HZ BOOKS
华章科技

技术领导力

程序员如何才能带团队

周明耀◎著

海康威视资深技术专家10余年团队管理经验总结，为程序员晋升管理者提供能力模型和进化路线图

从技术管理工作内涵、技术团队管理、产品开发过程管理、技术调研/预研、软件系统架构5个维度阐述技术管理者需要具备的能力

TECHNICAL LEADERSHIP

FROM PROGRAMMER TO LEADER

机械工业出版社
China Machine Press

内容简介

程序员不会带团队，就只能一辈子写代码，虽然写代码并没有什么不好，但是大多数程序员不愿意这样过一生。

程序员要带团队，要成为技术团队的领导者，必须在技术和管理两个方面有所长。技术方面，要以CTO为榜样；管理方面，则应该像CEO一样思考。具体来讲，要成为技术团队的领导者，要具备多项综合性的能力，如：

- **技术开发能力**：熟悉各种主流开发技术，精通其中部分关键技术；
- **项目管理能力**：能主导和管理项目开发的全流程，并应对过程中发生的各种突发情况；
- **产品研发能力**：熟悉产品研发的生命周期管理；
- **技术选型能力**：能正确地对新技术方案进行调研和预研；
- **系统架构能力**：掌握系统的软件架构方法论，熟悉各种常见软件系统的架构与设计方法；
- **团队管理能力**：能正确地识人、用人。

……

本书结合作者10余年的技术团队管理经验，从一线实践角度告诉你如何带领软件研发团队，如何才能从程序员转型为技术团队管理者。内容涉及技术管理工作、团队创建及人员管理、产品开发过程管理、技术调研/预研、系统架构基础知识五个方面，帮助读者快速实现从程序员到管理者的转变。

技术领导力

程序员如何才能带团队

周明耀◎著

TECHNICAL LEADERSHIP

FROM PROGRAMMER TO LEADER

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

技术领导力：程序员如何才能带团队 / 周明耀著 . —北京：机械工业出版社，2018.1

ISBN 978-7-111-58914-3

I. 技… II. 周… III. 企业管理 - 技术管理 - 研究 IV. F273.1

中国版本图书馆 CIP 数据核字（2017）第 329724 号

技术领导力：程序员如何才能带团队

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：李 艺　　责任校对：殷 虹

印　　刷：北京瑞德印刷有限公司　　版　　次：2018 年 1 月第 1 版第 1 次印刷

开　　本：170mm × 230mm 1/16　　印　　张：17.25

书　　号：ISBN 978-7-111-58914-3　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

序 1

Michael（周明耀）是我的好兄弟，我们已经认识 10 多年了。这些年来，我已经渐渐远离 IT 技术管理。而 Michael，一直坚持在 IT 技术管理的领域，以更加丰富的工作经历刷新、实践和升华自己对这项工作的认知和理解。

学习 IT 的人是比较容易上手技术管理的，但要做个好的领导者不容易。我们一定都学习过项目管理方面的课程，而且都做过项目，这可以看作技术管理最初的实践——无论是项目的参与者，还是负责人，至少我们都对团队，对管理流程（规则）、项目过程和项目结果有直接的体验。但是，要做领导者，面对的不仅仅是技术问题和解决方案，还需要面对上级、下属、同级别同事、客户等各种各样的人群和各种各样的事件，这时候只有高智商和强逻辑性就远远不够了。要成为一个真正好的技术领导者，除了热爱技术、功底深厚之外，还需要不断提升眼光（眼界）和决策能力，修炼勇敢、真诚、公平、开放、包容的态度，修养个人魅力。从组建团队开始，以人为本，从心出发，平衡“管”和“理”，并从中探索自己的管理哲学、方法论和行为准则。Michael 的书会给喜欢技术领导力的你们带来鲜活的自身经历、丰富详实的案例和恰当的建议。

读 Michael 的书很轻松，他将自己的故事娓娓道来，且处处不

离技术领导力的主旨。他以对电视剧中主要人物的技术管理角色分析作为开篇，生动形象，一开头就深深吸引了我。

希望你们也和我一样喜欢这本书，并从中受益！

TCL 医疗集团产品中心总经理　贺军

序2

用管理的方式做技术

我曾经思考过一个问题：“技术人员最大的发展瓶颈是什么？”当时我认为是技术方面入门容易，但深入很难，或许加强技术深度就是技术人员最大的挑战。随着工作经验的积累，做的东西多了，接触的面也广了，加上对技术一直抱有由衷的激情，自己在技术深度与广度方面都有了明显的提升，也形成了一定的质变。但是，此时我更加迷茫了，到底应该如何走好自己接下来的技术之路呢？继续将技术深入下去，还是从技术转到管理？也许我曾经面临的这个问题，大家目前正在经历中。

技术就像无底洞，我们在洞口目测该洞可能只有 10 米深，但当我们走到 10 米深处时，却发现可能还有 100 米才能走到底，而且越往深处走，越感觉深不见底。如果感觉自己已经走到了最深处，那里可能没人了，但某天也许会发现，还有更多的人比我们走得更深。

技术就像武功，人外有人，天外有天；但技术也不全像武功，它依靠的不是单打独斗，而是团队协作。当我们认为自己的技术已经达到了期望的深度时，应该选择更加灵活的方式来提升自己的“武功”，而不是一味地想让自己成为天下第一，因为根本就没有天下第一。此时我们更需要做的是，通过自己的管理技能去带领更多

的技术人员，通过团队协作来取代单打独斗，用管理的方式做技术。

也许大家和我一样，选择了走技术管理的路线，去攀登更高的技术山峰。在攀登技术之巅的路途中，我们或许犯过类似的错误。

1）队员工作没有自己做得好，花时间去沟通还不如自己去做。

2）感觉自己的技术在退化，长期不写代码觉得自己没有价值。

3）发现自己不懂的东西越来越多，觉得把管理做好不太容易。

当大家遇到以上问题时，无须过多担忧和自责，更不要认为自己不适合走技术管理路线。其实以上问题只是一个信号，它在提醒大家，是时候提升自己的“软技能”了。

如果将技术能力视为“硬技能”，那么管理能力就是“软技能”，而软技能所涉及的面可能比硬技能更广，甚至更为抽象，其中很多软技能考验的不是单一的能力，而是综合的素质，比如素养、胸怀、情商、智慧等。有些方面可能受自己的成长环境所影响，但更多的软技能其实是可以训练的，而且完全可以通过正确的方法得到明显提高。

我原本以为周明耀老师只是一位狂热的技术爱好者，在 JVM 和 Java 性能优化方面达到了登峰造极的境界。前不久听说他在写关于技术管理方面的书，我当时还真有些惊讶，但当我看到这本书的部分内容时，我心中的惊讶消除了，而且全部变为期待了。能在技术与管理方面有如此造诣的人实在不多，能够将自己的经验总结下来并分享出去的人就更少了，周老师令我十分敬佩。

强烈向大家推荐这本书，因为它是一本技术高手的管理心经！

黄勇，特赞科技 CTO

序3

管理是一种实践，其本质不在于“知”而在于“行”。(Management is practice. Its essence is not knowing but doing.)

——彼得·德鲁克（Peter F. Drucker）

对于多数勤勉的工程师来说，努力数年后走上技术管理岗位似乎是一件水到渠成的事，但要真正做好技术管理，成为一名优秀的管理者却并不太容易。认识明耀已有数年，知道他是一名优秀的软件开发团队管理者，也拜读过他的一些技术著作，但当听到他要写一本关于技术管理的书时，我还是有些吃惊。不同于一般的项目管理和职能管理，技术管理的主体对象是最具特色的知识型员工，具有很大的不确定性。每一种行业、每一个技术领域、每一家公司，甚至每一个技术团队，都有可能采用不一样的管理方式，我很好奇他想从哪些方面进行提炼。

几番闲谈之后，我基本明白了他写作的初衷。作为一名有着近十年技术管理经验的IT技术狂热爱好者，他一直想对自己的工作内容、经历进行总结，并且将自己的一些技术管理经验和理解分享给大家。本书从技术管理者的基本素质要求谈起，以软件开发工作中的实例来介绍他对技术团队建设、人员管理、产品开发过程管理的一些理解，并且重点介绍了作为技术管理者的另一个身份——技

术总体负责人在技术调研、预研和系统架构设计方面的一些经验体会。非常欣赏他这种勇于总结实践经验的做法。管理的本质在于“行”，写书的过程不也正是提升管理的一种修行吗？

本书可供有志从技术走向技术管理岗位的产品开发、测试人员阅读，也可供相关领域的技术管理者共勉，当然也可供对技术管理领域感兴趣的所有人员阅读。

高凌　浙江大学 MBA、PMP，职业经理人

序 4

技术领导之路

关于技术领导，有人把这个概念定义成一种职位，比如技术经理、技术总监。有人把它引申为技术上的洞察和优势。我则认为，如果你有能力把技术相关的资源有效地组织起来并完成一件有价值的事情，例如，发布一款产品，做一个项目，那么你就是技术领导者，或者说，你具备了一定的技术领导力。

以前有很多人问过我类似的问题：单纯做技术，核心竞争力就是技术强，简单明了，做技术领导感觉就没那么明了，哪些是技术领导的核心竞争力？是对系统的整体理解，组织协调能力，个人品质素养，还是技术领导力本身？从职业发展角度看，技术领导最终会晋升为某个管理职位，而管理职位应该是一个更难胜任的职位才对。技术弱化带来的危机感（不安）的本质原因是什么呢？

我觉得周明耀老师的这本书很好地回答了这些问题。

简单来说，技术领导可能是个管理岗位，也可能不是，不重要。重要的是你要利用技术资源和团队资源把事情做成。从技术人员和技术领导的分布上来说，后者显然更难胜任。从分工上看，也不可能有那么多领导者。技术领导者的核心竞争力应该包括但不限于：技术能力，对事情整体的理解，能找到正确的方向，影响力，凝聚力，对人性的理解，资源。技术弱化为什么会带来的危机感

呢？因为人们总觉得要有一技之长才会比较安全，但是优秀的技术领导会超越这些东西，他们关注的内容不再是某个具体的技术和实现，而是事情。让正确的事情，持续发生才是最重要的。

技术领导者应该具备什么样的素质呢？书中给出了这样的答案：

技术，技术，技术

一旦技术人成长为技术领导之后，有个问题就会像“我是谁”一样一直困扰着我们：我还需要在技术领域孜孜以求吗？答案当然是要。你是技术领导啊，又不是产品经理。技术这东西是很实在的，泾渭分明，会就是会，懂就是懂，很难不懂装懂。在现在这个时代，技术是需要我们终其一生学习的东西。

管理看起来套路很多，其实最终都是人性和策略，理解人性，善用策略，就能做好管理。对于聪明人来说，有实践机会，管理可以在短时间内达到一个不错的水准，但技术永远需要长时间积累。钻研技术，并不是让你增加自己的代码量，事实上一个领导者每天深夜像打字机一样咔咔地提交代码，对组内成员是极大的压力。一个技术领导，更多是通过对技术领域的探求打磨自己的技术敏感度和技术决策力。

如何用好当下的技术解决现实中的问题，什么阶段引入什么技术，什么时候重构，什么时候重写，如何利用技术驱动产品，如何构建技术平台……这些都是技术领导需要思考并确定的问题，这些都将依托你强大的技术背景。

信任

相信自己的团队，就能产生巨大的生产力。事实上，如果你选对了人，大部分看起来困难的事，都可以解决。

很多时候，团队的人跑过来问你怎么办，只是希望你给他们信心，而不是指望你去给他们写代码。除了需要资源协助的情况，大部分时候你只需要信任他们，

然后等着他们告诉你，问题已经解决了，系统已经上线了，产品已经发布了。

程序员对技术的渴求和敏感度，就像枝桠对阳光和雨露一样渴望和迫不及待，只要等，大部分时候，他们都能找到出口。当然，真的遇到困难搞不定了，协调资源或自己提刀上阵就是了。

鼓励和批评

把鼓励和批评放在一起说，因为它们是一对双刃剑。无节制的鼓励和表扬会导致你成为一个烂好人，而随时随地的批评会打击团队的自信心，人心离散，智慧之光凋零。如何取舍呢？你需要找到自己的平衡。

有的人喜欢多鼓励，少批评。在平时的交流和会议中，多给予鼓励和表扬，效果有时候比正式会议的褒奖更让人感觉舒适。少批评，但批评的时候一定是声色俱厉、毫不留情。有的人则相反，少有表扬，多为批评。时时严厉的人，偶尔一次褒奖，会让团队成员觉得如饮甘露，有时候效果也非常好。

找到适合自己团队的方式就是最好的方式。

团队作战

很多技术领导带团队取得了一点成绩，就开始沾沾自喜，以为这事离了自己不行，其实是团队作战的功劳。大部分情况下，不是团队离不开领导，而是你离不开你的团队！

你是团队的头，但你的技术不一定是最强的，而你要依靠他们的努力工作来开发出强壮、稳定的软件。你的工作是什么呢，除了技术，还需要保护团队中的工程师不被打扰，在各方面给他们支持，帮助他们能顺利完成任务。同时你还需要在重大的方向上输出影响力，做决策，定计划，这也意味着，如果决策是错误的，需要你来承担责任。而有了荣誉的时候，你要退下来，让给团队里那些优秀的人。这样的团队，才能无往不胜！

善用人才

稍具规模的团队，人才都会有很多种，有的是匠人型，有的是天才型，他们有的人步伐齐整，一步一个脚印，行走就像时间本身一样流畅，唰，唰，不可阻挡。有的人似乎原地不动，呆呆思考，但是瞬间会来个百米冲刺，把所有人都落在身后。

无论是行走，还是奔跑，他们都需要合适的道路和轨迹。设计好这样的通道，帮助这些人才成长，他们自然会做出成就。

我写的这些只是书中内容的冰山一角。本书的作者近十年来一直从事研发团队的管理工作，在研发一线摸爬滚打，积累了大量的相关经验，同时热爱分享和技术写作。他曾经著有《大话 Java 性能优化》和《深入理解 JVM&G1 GC》，本书是他的第三本书。在书中明耀用轻快的笔触和充满节奏感的文字，从技术管理、团队建设、产品开发过程、技术预研和选型、系统架构等各个层面，带领你游历属于自己的技术领导之路。

如果你是一位技术从业者，如果你想打造自己的技术领导力，本书就是你手边的参考手册。而我的责任，就是尽可能向更多的人推荐这本书。

池建强

极客邦科技总裁，MacTalk 出品人

前言

为什么要写这本书

我对于“技术管理”这个词，或者这个岗位的理解是，先有技术，才有管理，所以，我先写了两本技术类的书，分别是《大话Java性能优化》和《深入理解JVM&G1 GC》，谈了谈对于技术的理解。现在我觉得可以谈谈对于技术团队管理的理解了。

近10年来我一直从事技术团队管理工作，一直对自己说要站在一线。我认为所有的工作都应该有对应的方法论支撑，技术管理工作也不例外。当初次担任经理的职务时，我也是无限惶恐，于是上网找资料、看书，有一次看到一篇博文说“经理的职责是管人，而不是管事”，支持这种论调的人其实不在少数，但在思考、实践了几年后，我觉得这种说法真的不适用于科技行业的技术管理领域。鉴于目前市面上几乎没有真正介绍如果做一位合格的技术管理经理（Leader）的书，所以我觉得我应该对所做的工作做一次深层次的总结。

技术管理岗位不容易做，既要保证自己的技术说服力，又要经常上一线工作，还要从管理上给予团队支撑，这些综合起来对于任何一位工程师来说都不是那么容易理解和执行的，希望我的这些总结能够真正帮到你，让你觉得这是一本有意思、有价值的书。

本书主要特色

希望大家读完本书之后，能够大体上解答你心中对于如何才能更好地带领软件团队的诸多疑惑。这里有几点需要说明。首先，本书总结的方法论是我个人的总结，力求实践，不会多讲空话、套话，但也可能存在自我认知的局限，请读者见谅，多提意见。其次，我觉得人脑是越学越灵活的，也许 5 年后我会认为现在的方法论存在一些问题，到时候我会写文章自我纠正。再者，技术管理人员很多时候就是一个小号的 CTO，如果需要自己决定产品方向、销售，那就是个小号的 CEO，所以技术领域的综合能力也是你的必备能力。最后，我肯定会继续站在技术管理一线，无论 Title、薪资、职场斗争，哪种都不会让我改变自己的职场定位，与君共勉。我相信，从软件工程师转变为技术管理者的你，一定可以从本书中看到自己成长的影子。

我觉得，技术管理本身是有方法论的，但是技术管理人员很多不能得到别人的理解。希望通过本书的总结，帮助大家更好地了解技术管理工作，也更好地做好技术管理工作。

本书读者对象

根据本书的内容特点，可以将阅读对象分为以下几类：

- 尚未走出校门，但是心怀远大理想的计算机、软件专业的学生；
- 一线研发工程师；
- 工作几年后遇到瓶颈的工程师；
- 刚刚担任小组长的工程师；
- 有几年带人经验的研发经理；
- 对技术管理岗位有兴趣的同仁。

如何阅读本书

本书总共 5 章，5 个章节内容相互独立。分别从技术管理工作概述、如何进

行团队创建及人员管理、产品开发过程管理、技术调研 / 预研工作介绍、系统架构相关知识介绍如何具备技术领导能力。每一章内容相互独立，你可以按顺序阅读，也可以选择选择感兴趣的章节阅读。

勘误和资源

由于本人水平有限，编写时间仓促，书中难免有错误或者不全面的地方，在此恳请读者朋友批评指正。如果你有任何疑问或者问题，可以通过我的微信号“ michael_tec”与我联系，也欢迎关注我的微信公众号“麦克叔叔每晚 10 点说”，期待你们真挚的意见反馈。

致谢

7 岁那年，当我合上《上下五千年》时，我对自己说，我想当个作家。2016 年 4 月我出版了《大话 Java 性能优化》，1 年内加印两次的销量让我很满足，因为有读者购买我的书，是对我的支持，我需要继续坚持写作。2017 年 5 月，我出版了《深入理解 JVM&GIGC》，现在，我的第三本书也即将面世。坦率地说，本书更像是我对过去 20 多年学习、工作的回顾，是一本自述题材的管理类书籍，写作过程也几乎是一气呵成，希望能给读者朋友帮助。

对于我的每一本书，我都怀着忐忑、惊喜的心情，就像第一次面对我的女儿“小顽子”，给她取这个小名，希望她顽强到底，因为我相信，你若顽强到底，一切皆有可能。

每次遇到困难时，耳边总是回响着郑克良老师（杭州高级中学退休老师，浙江省第一批特级化学老师）的那句鼓励“永远不要放弃”，人应该学会回馈社会，技术分享也是回馈社会的一种方式，而以写作方式进行技术分享是我所能最大程度回馈社会的一种途径。

感谢机械工业出版社华章公司的杨福川老师和李艺老师，你们的专业和热情深深打动了我，让我能够坚持完成本书的编写。同时感谢你们对本书页面排版等

内容的包容和支持。

感谢 TCL 医疗集团产品中心总经理贺军、极客邦科技总裁池建强、特赞 CTO 黄勇、职业经理人高凌等为我写序。

感谢我的家人，是温暖和谐的家庭帮助我完成了这本书。我的妻子美丽、细心、博学，虽然偶尔不那么温柔，但是我很爱她。我的小顽子，她天生性格很像我，希望她能够踏踏实实做人，保持创新精神，平平安安、健健康康地生活下去。感谢我的岳父母、我的父母，感谢他们帮我照顾小孩，让我有时间编写此书。感谢浙江省特级教师、杭州高级中学（杭一中）化学老师郑克良老师，郑老师的一句“永远不要放弃”，是我多年前进的动力。感谢数学老师张老师在公开场合对我智商的褒奖，第一次收获这样的赞赏，对我这样性格内向、内心细腻的孩子是非常重要的。感谢生命中出现的恩师、良友，有你们的存在，让我得以绽放。

谨以此书献给记忆中的爷爷奶奶、外公外婆，你们给我的都是最美的回忆，我永远爱你们。

我相信这本书不是终点，它是麦克叔叔此生一系列技术书籍的其中一员，欢迎加入我的朋友圈，可以通过微信号 michael_tec 或者微信公众号“麦克叔叔每晚 10 点说”和我联系，咱们下一本书见。

Write great book that millions could read is my passion!

周明耀

写于杭州高级中学（杭一中）贡院校区

目录

1

第 1 章

技术管理工作

法国著名作家雨果 26 岁时已经出版了 10 本作品全集，除了写作天赋受人尊敬以外，他的高尚品德同样令人钦佩。在英法联军火烧圆明园后，他发表了那封著名的抗议信；巴黎公社失败后，他呼吁赦免公社战士，并且开放自己在比利时布鲁塞尔的住所供社员们避难；从 1862 年起，他每周固定招待 40 个穷孩子吃一顿晚饭，等等。虽然福楼拜先生承认自己和雨果先生的世界观不同，但是仍然一直坚持等待，直到雨果的巨著《悲惨世界》出版以后才发表自己的小说《萨朗波》，这是因为福楼拜认为，雨果是那种在他面前你需要欠身相让的人，因为他的工作能力和他的个人魅力。作为技术团队管理者，如果我们可以得到员工，甚至对手发自内心的尊重，那才是真正的成功，而不是用金钱和职权所能够计算的。

近 10 年来我一直从事研发团队的管理工作，一直对自己说要站在研发一线，以此保持自己的技术能力，同时不断加强技术管理能力，因而有机会沉淀了一些技术管理方面的经验和个人理解。我认为所有的工作都应该有相应的哲学思想，或者至少有方法论的支撑，技术管理工作也不例外。当初次担任技术团队管理者的职务时，我也是无限惶恐，工作当中也曾遇到很多困难，也用糟糕的方式给自己制造了一些麻烦，于是上网找资料、买书。有一次看到一篇博文说：“团队管

理者的职责是管人，而不是管事”，并且发现这种论调其实不在少数。但在思考、实践了几年后，我觉得这种想法真的不适用于科技行业的研发团队技术管理。也正是由于这些年没有看到较为全面地对一线团队的技术管理工作进行讨论和总结的书。所以，我决定写一本。

本章主要介绍和解决以下问题，这些也是全书的基础：

- ❑ 什么是技术管理，为什么这么难做。
- ❑ 技术管理包含哪些方面。
- ❑ 如何做一名合格的团队领导者。
- ❑ 团队领导者应该具备哪些品质。
- ❑ 带领技术团队的心得分享。
- ❑ 个人职业发展需注意的事项。

1.1　技术管理

上面这张图是位于钱江新城的杭州图书馆，该馆是国内最具互联网思维的图书馆，并不是指它的硬件，而是软件（管理思维、方式）。硬件容易模仿，软件则需要自上而下的认知。继多年前对拾荒者、无家可归者开放后，在 2016 年，杭州图书馆又将书的选择权出让给了读者。读者可以持杭州市民卡在新华书店任意借 N 本新书，通过读者试读反馈的信息确定图书馆每年的最终采购图书书目。更深层次地理解，一本书摆在书店里，是未经筛选的，而摆在图书馆里，则是筛选过的。杭州图书馆把选书的权利交给了读者，使每位读者都有一定的参与感和成就感。读者知道这本书以后可能会有很多人看，这是他在为其他读者选书，所以一定会尽量给出客观公正和全面的评价。这是一个很好的共享。

在 IT 业内，无论是互联网产品，还是成熟的企业级应用，或者是手机、平板电脑，他们背后都有一个团队，所以个人的高效能不可怕，团队的高效能才可怕，当团队中每个人的潜力都被发掘出来，能力的集合才是巨大的，才能所向披靡。技术管理就是这么一种组织团队共同进行研发的工作，它是一门细分工种，随着社会的进步、人类思维的开放、经济水平的提升，被管理者（工程师们）逐渐从单纯为了解决物质矛盾转化为追求工作的归属感、参与感、技术尊重感，他

们逐渐对跟着谁一起工作、采用哪种工作方式、解决了什么样的技术问题、做出了什么样的产品感兴趣，而不是仅仅满足于金钱的多少。同时，被管理者之间还存在紧密的社交关系，能够随时保持沟通，分享对在该公司或组织工作的切身感受，分享对于管理者个人能力、品德的认识。我们每位技术团队管理者，也正在像书一样被摆放在被管理者面前，如果别人懒得翻你，就证明你不能给予他们指导，他们没有归属感，最终都会匆匆离职。

好了，可以开始我们整本书的内容了。我认为，无论工作或是生活，最好的情况是能够做自己喜欢的事情，能够跟自己喜欢的一切在一起。技术管理是我所喜欢做的工作，只要还有我的生存空间，我会一直做下去。

1.1.1 程序员的名义

前段时间，电视剧《人民的名义》很火。电视剧中角色较多，下面我将基于个人专业角度，分析哪些角色符合技术管理者的要求，分析不一定准确，也可能不够全面，请读者见谅。

李达康

1）遇事急躁。高小琴向他告状有工人阻拦厂区搬迁，他不过问什么原因，就认为是老百姓的问题。这件事充分体现了他遇事不够冷静、思考不周的性格。

2）做事直奔目标，抓小放大。厂区发生大火，老检察长好不容易平息工人的愤怒，他却在祈同伟的“点拨”下想趁乱拆厂房，而没有仔细研究为什么工人要造反。

3）不擅用人。副市长贪污腐败严重，还整天在外面说是“达康书记的化身”；区长整天看星星，不干实事。对于这两位下属，他并没有第一时间察觉问题。

达康书记的优点和缺点都过于鲜明，不宜在关键部门贸然担任技术管理者，需要时刻保持观察，或者从基层一步步锻炼，且不宜提拔过快。一旦提拔为正职，最好能够配一名类似季昌明或陈岩石这样的同志作为主要副手，辅佐、监督其工作。

沙瑞金

1）刚上任即到一线调研，且时间较长，充分体现了他深刻理解“从人民中来，到人民中去”的道理，没有脱离一线。

2）接到高育良请示如何处理丁义珍的电话时，他只回复：“我才刚到任，还不熟悉情况，你们酌情处理”。处理很得当，既没有过多干涉自己不了解的事情，也把棘手的皮球巧妙地踢开，为之后的介入、接盘留下空间。

3）看新闻播报工厂强拆引起大火事件时，立即放下早餐认真做笔记，为后续常委会讨论内容准备大方向。

4）处理工厂强拆引起火灾事件，打电话指示李达康首先做好人员抢救、安置工作，把拆迁放一边去。处理逻辑合理、思路清晰，做到了抓大放小。

5）发现基层公务员当中的务实者没有晋升空间，认真思考对策，帮他们搭建平台，进行提拔。

如果有这样的人出任 CTO 或研发副总裁等研发类高管职位，那是研发人员的幸运，他会真正理解技术人员，他所带领的团队一定积极向上、热爱技术、处事公平，一定会在科技圈交出满意的技术答卷。

陈岩石

陈老干了 20 多年副检察长。这样的人一般技术过硬，为人正直，出任研发总监职位，团队里的人不敢不用心做事。真正干技术的人，会很喜欢这样的领导，从他身上能学习很多技术本领。我最近就遇到了这么一位高管，35 岁教授级工程师，39 岁副所长，45 岁副总裁，用技术领导力训下属，下属会心服口服。

季昌明

如果没有季昌明的暗中安排、支持，其实沙书记很难完全应付高育良，侯亮平和陈海也很难开展实际工作。他很稳重，考虑问题很全面，如果季昌明这样的人出任技术高管，或者对应职位的副手，那么底层技术人员就有可信赖的沟通渠道，可以顺畅地与上层沟通。

侯亮平

最高检人民检察官，为人公正、公平、正义，一身正气，他来到汉东省担任反贪局局长，就像一只孙猴子一样，开始搅浑汉东的局势。这个角色有胆有识，可以重点培养为技术管理者，且遇事分析思路清晰、不按照常理出牌，适合做创新团队的技术带头人，在他的带领下，团队应该能够攻克技术难点，为公司带来新的技术创新点、产品赢利点。

高育良

这类人城府很深，做事能力极强，但是内心有着极强的私欲，且很会利用各种各样的资源，在大型企业中肯定会有。对于这类人，用得好则对公司极大有利，用不好则会导致风气败坏、拉帮结派。

祈同伟

这类人是团队的蛀虫，是规则的破坏者，他们为一己私利可以做任何事情，所以不适合做技术管理者，最好能够彻底清除。

孙连城

这类人挺多的，拨拨动动、毫无主动工作能力，技术技能也大多比较落后，让技术团队管理者很无奈。作为他们的领导，应该多和他们正面沟通，带动他们的工作积极性，让他们明白，技术团队管理者也是打工的，也是拿工资的，也要养家，谁都不能不劳而获，免得被洗牌时抱怨："老板就是资本家，我的领导是帮凶。"

总结

沙瑞金、陈岩石、季昌明，他们这三位都适合做技术团队管理者，都是管理层的骨干成员，对于技术团队的把控一般来说不会出现问题。

侯亮平和陈海，属于可以重点培养的团队骨干成员，应该让他们开始独立带人，慢慢积累经验，定期与他们交流，给予他们指点，帮助他们成长为能力全面

型队员。

李达康这类人做事情风风火火，团队管理上很霸道，如果需要团队短期内出成绩，可以使用这类人，但是如果你看重的是长期行为，那么需要让经验丰富的 HRBR 辅助他工作，避免出现“一言堂”的情况。

高育良这类人进入了管理层后，需要保持和他们的沟通，积极引导他们向正面前进，避免他们自搞小团体。

祈同伟、孙连城这两类人，都不适合现代企业的技术管理。

1.1.2　为什么技术管理难做

我们应该都有听过某互联网公司爆发的所谓“ CTO 是否需要写代码”的争论事件。公司高层列举这个人怎么没用，第一条就是这个人居然几年时间没有写代码。我不评论双方对错，不评论这样的指责是否属实，也不评论他是否一定需要写代码，我只是觉得，如果是一家成熟的科技公司，它应该从多个维度评审技术团队管理者的工作过程和成绩，而不是采用单一化规则进行评判。

坦白说，技术团队管理者这个工作，想做得让公司内大多数人满意，难度非常高。前不久一位网友向我诉苦，说他的团队管理者只会管进度，其他什么也帮不上。这类抱怨很多，很多人都在抱怨他的领导无能，进而大家都觉得技术管理是一个很虚的工作，没有技术专家那么实在。造成这样的误解，我认为主要原因是技术管理者的需求方太多，接下来我们以假设人物举例说明。

1）刚通过校招进入团队的 A

A：我有一个函数不会写，我使用 Spring 框架时老是报错，能不能帮帮我？

2）团队内工作 3 ～ 4 年的 B

B：我们对外的协议层性能不好，不知道怎么优化，能不能指点我？

3）团队内工作 5 ～ 6 年的小组长 C

C：系统架构能不能给我点意见？技术选型怎么做？

4）产品团队管理者 D

D：技术要符合产品需求，别谈什么技术追求，让你们做什么产品就先做什么。

5）直属领导 E、F

E、F：你的编码能力怎么样，系统架构能力怎么样，项目管理能力怎么样，项目不要有任务延期，未来技术方向定出来，设计文档怎么写得这么差，对外沟通工作怎么做得不及时……

6）部门最高领导 G

G：我们和同行、业界的技术差异点有哪些？为什么你们不用那个流行的某某框架？

7）兄弟部门领导 H

H：你要站得更高一些，能不能讲讲未来几年云计算的发展趋势？

8）文档评委 I

I：你们这个协议定得不够好啊。

9）HR 接口人 J

J：团队氛围很重要，公司价值观要注入。记得要给新员工进行技术培训。

10）专利部门接口人 K

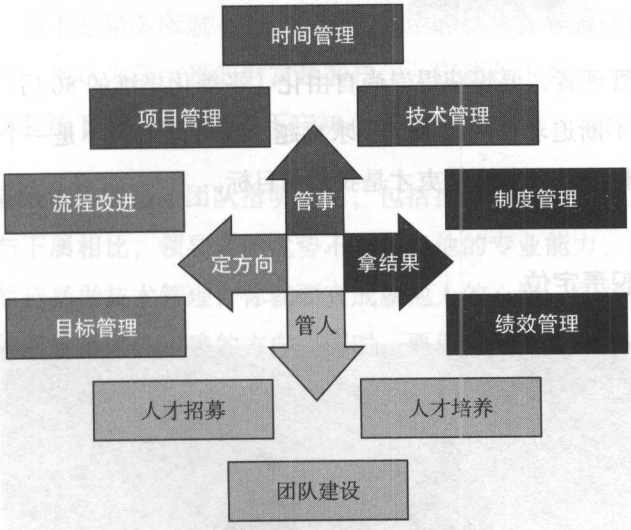
K：专利也很重要。有没有开始专利布局？

综合以上这些需求（实际还不止这些需求来源方），作为技术团队管理者，无论具体管几个人，最好能够拥有以下能力，才能满足各个需求方提出的需求：

- 深入理解一门或多门编程语言

- ❑ 深入理解多种流行的框架
- ❑ 系统架构能力强，拥有复杂系统的设计经验
- ❑ 积极跟随开源社区
- ❑ 积极了解业界技术发展
- ❑ 沟通能力强、情商高
- ❑ 有产品意识，不是技术迷
- ❑ 会带人，服从领导，责任心强
- ❑ 会写专利
- ❑ 再会点别的更好
- ❑ 随叫随到、工资不高

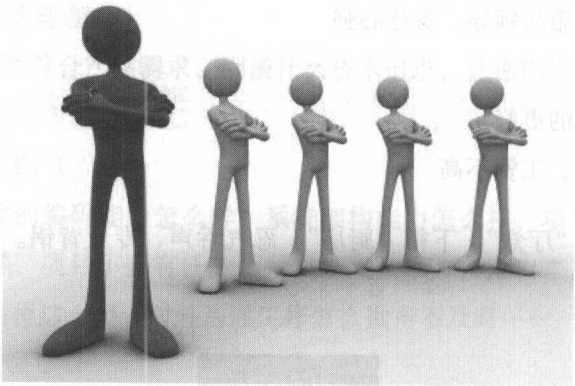
总之，上得“厅堂”、下得“厨房”、忍气吞声、专业背锅。这些可以用下图展示。



可见，技术管理真不是那么简单的，作为技术管理者，需要持续保持自己的技术能力，需要不断强化自己的管理能力和方方面面的综合性能力。此外，还要遵守技术管理者的职业素养要求，这是因为，职业生涯本身就像是在一个棋盘上下棋，既然选择了技术管理这条路线，就要遵从它自身的约束和要求。

记住，所有的失败都是有利于我们成长的。既然选择走上技术管理道路，无论前途是否艰难，都要咬着牙走下去，只要别人不赶你，你就要做下去。

1.2 技术团队领导者



作为技术管理者，要想获得崇尚自由化、平等化思维的 80 后、90 后的尊重，我认为，唯有不断追求自我提升、追求卓越。狡臣酷吏绝不是一个好的领导，也不应该启用这样的人，良臣干吏才是我们的目标。

1.2.1 工作职责定位

一家企业能够生存，一定是业务开展得不错，作为技术团队管理者我们需要做到技术与业务的融合，需要保证我们的付出可以为企业带来长久的利润。只有成为“技术的业务派，业务的技术派”，才能让技术引领业务，并创造价值。这其实也是技术管理者成熟的标志。

以电商企业为例，如果技术团队的定位仅仅是为了完成业务部门提出的需求，那么对公司的价值是不大的，技术团队的定位应该是“技术引领业务”。首先，技术人员要成为业务专家，能够通过系统性的思考，规划出跟企业管理思想高度一致的电商系统。同时，系统必须具备柔性，因为电商的模式是多样化的。只有这样，系统才能成为业绩增长的基石，成为企业的核心竞争力，为企业提供强大的技术保障。

2017 年 4 月 18 日，任正非在华为战略预备队座谈会上进行了一次演讲，依然是以“敲打”为主，他以美联航驱逐乘客事件为例，告诫华为员工要时刻坚持以客户为中心，而不是像美联航一样，以员工为中心，从而导致他们对客户这样恶劣的经营作风。他认为最宝贵的财富是客户，一定要尊重客户。华为以客户为中心的文化，要坚持下去，越富越要不忘初心。

一位管理者最重要的是给团队指明方向，包括技术方向和业务方向，还有个人成长方向。与下属相比，领导者的优势不只在于他的专业能力，还在于他的眼光和胸怀。既然选择做技术管理，你就要有成就他人的心胸，同时还要有超出一般人的眼界，能够给予团队正确的方向。同时，要保持对技术的兴趣，多关注新技术的发展，否则你很难在重大的技术决策上做出正确的决定。

此外，作为一名技术团队管理者，有很多内容需要注意，其中有一条是需要放在第一位的，只有做好这一条，你才能够让团队感觉到公平。

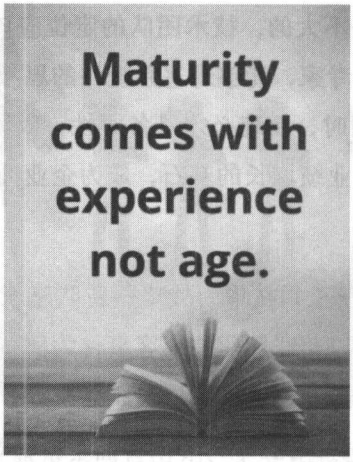
“你要维持一种正义和公平，这是非常重要的，不要欺负弱者。当你是强者的时候你欺负了他，他强了他一定会欺负你。有一种感恩的心理很重要，会把事情做得很好，否则没法维持”。

1.2.2 工作品质定位

1. 成熟

“走向成熟就是独立得更彻底而又联系得更紧密。”

——霍夫曼·斯塔尔



成熟的人具备一些典型特征，他们重视诺言、不夸夸其谈、有学识而含蓄内敛、心胸宽广、不以自我为中心、勇于承认错误、意志坚定。凭自身的品格、实力与信誉在人群中从容地来去，这样的人流溢着非凡的魅力。成熟的人刚强豁达，待人宽容、知足乐观，他们知道为人处世的原则，懂得如何处理好与朋友、同事的关系。

技术团队内部有一些员工，也许他们不善言辞，但交付的任务，他们一定会尽全力完成，遇到不懂的地方也会主动请示、沟通，个人姿态摆得很低，这样的员工内心非常成熟，知道如何做事、体谅上级、努力工作，而不是把精力放在正经工作以外的地方。他们是团队的中坚力量，是不可或缺的成员。我认为衡量一个团队的凝聚力是否强大，看看这类员工的占比就知道了，缺少这类员工的团队一定不会有什么了不起的成绩。

每天回到家，我开始履行我的另一份职责，洗衣服、陪孩子学习、陪家人聊

天、写作。这就是生活，一个男人结婚生子后的生活一定是与单身时不一样的。不要觉得这不是男人干的活（现在已经不是男权时代了，家庭责任需要一起承担），要能理解生活，主动且乐于承担生活的重担。

坦率地说，作为一名技术面试官，我一直对较为成熟的、能吃苦的、能理解别人的人情有独钟。身在职场，我们都需要可靠的团队成员一起帮扶前进。任何不成熟的人，最终都会在职场上流露出他的自私、明哲保身、浮躁、乱说话、懦弱等品质，所以一些自认为能力很强，认为自己不受重用、永远在底层的员工，在埋怨领导之前，应该多多自我反省，是不是工作职责没有承担起来，是不是自己为人处世的方式让别人很难堪而不自知，是不是开口闭口都是以自我为中心。

2. 勇敢

“我认为克服恐惧最好的办法理应是：面对内心所恐惧的事情，勇往直前地去做，直到成功为止。”

——罗斯福

80 年代出生的男生应该都看过《灌篮高手》这部动画片，一支技艺一般的球队，依靠着教练、队长的坚定信念，逐渐吸引了一些优秀球员加入，并在缺少板凳深度的情况下，杀入全国总决赛，最终击败了最强对手。这部动画片，改变了那个时代很多男生的梦想，也让大家学会了勇敢、拼搏。

《家园防线》这部由杰森·斯坦森主演的电影，讲述的是一位卧底当毒贩的警官，查出毒贩后隐姓埋名，后来又为了保护女儿而与坏人正面交锋的故事。电影很热血，父亲很硬汉，建议大家观看。

回到我们的团队管理领域。作为团队负责人，我们也应该有硬汉本色，不能惧怕外在的威胁。团队成员都在看着你，你决不能懦弱，要勇敢。举个例子，很多时候并不是你想团队怎么发展它就怎么发展的，很有可能你的团队刚刚明确产品目标、技术目标，人员也根据这些目标刚招聘到位，正准备大干一场的时候，忽然听说另外一个团队正在做一模一样的事情。别以为这是笑话，很多大公司都

存在这类问题，多数原因在于缺少顶层设计（当然也可能是顶层有意设计的，目的是让两个团队竞争）。面对这类情况，你可以选择退缩，转向另外的方向，也可以选择混混日子，等着被解散，但无论哪种选择，你的团队都有可能瓦解。每个人都有自己的职业定位，想做的产品、技术没了，有能力的人就不跟你玩了。我个人更倾向的是，在领导层不明确哪个团队做的前提下，对内组织团队尽全力做下去，而且要做好，对外则勇于承担责任，敢于接受挑战。

在电视剧《长沙保卫战》中有一段对话，参谋长对薛岳将军说："你若下地狱，我绝不上天堂。"这也是我在团队遇到严重挑战的时候，对所有成员说的第一句话。想做成事情，你或者你的核心下属，必须勇于站出来。

3. 热爱技术

"科学所以叫作科学，正是因为它不承认偶像，不怕推翻过时的旧事物，很仔细地倾听实践和经验的呼声。"

——斯大林

2016 年"天猫双 11 全球狂欢节"，单日交易额定格在 1207 亿元，被阿里定义为整个社会走向"新零售、新制造、新金融、新技术、新资源"的起点。经过 8 年的锻炼，2016 年阿里技术创造了惊人的记录——每秒同时创建 17.5 万笔订单以及 1 秒钟同时完成 12 万笔支付，成就了全球范围内最为庞大且复杂的交易体系和交易规模。正如阿里 CTO 所说，如果没有对梦想的坚持，以及对实现梦想的不懈努力，阿里不会是现在的阿里。实现梦想需要有强大的技术实力作为基础。以计

算为例，双 11 如此庞大的计算，一切关于搜索、推荐、人工智能的“梦想”都需要计算平台的强有力支撑，阿里如果不打破传统 Hadoop 框架的藩篱，自主研发非常高效的离线和实时计算平台，那么用户在交易的过程中就不可能有现在的体验。

我在 IBM 开发者论坛发表了 20 多篇文章，有一次同事通过一篇文章搜索到了所有文章，他很吃惊，跑来和我说，只有非常喜欢技术的人才会这样做。是的，我很喜欢技术，所以我才会和喜欢技术的人相处得很好，才可以一直带领着软件开发团队。

说点自己的想法。我对团队的技术要求一直都是：“我们输出或者呈现给别人的技术能力，需要且必须是公司内部的技术权威之一，说是之一，是因为不能否定公司内部其他部门的技术能力。我们必须让别人知道我们是专家，我们团队很牛。如果被别人列举出我们团队技术不尽人意之处，我会认为这是对我人格的侮辱，只要我是这个团队的领导，我决不允许这种情况发生”。

但有一点需要注意，只有做好当前的事情，你才有资格谈技术理想。

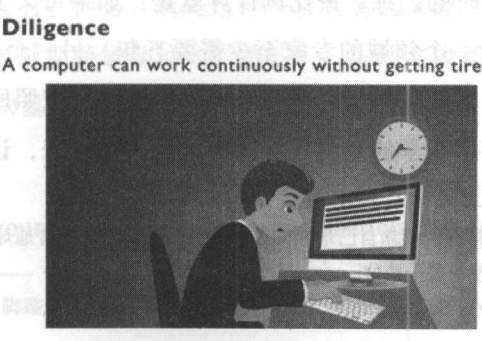
4. 勤奋

“如果你不喜欢改变，你会更不喜欢变得毫不相干。”

——Eric Shinseki 上将，美国陆军参谋长

“你们要像种子一样，到最需要的地方去，生根、发芽、开花、结果，再成片开成花海。部分勇敢的人，要到最艰苦的地方去快速成长。”

——任正非，华为总裁



最近在看加里·内维尔（曼联贝克汉姆时代的巨星，英格兰国家队主力右后卫）的自传，他自述自己是一个资质平庸的人，甚至比不上自己的亲弟弟，但是他最终通过自己的努力实现了职业辉煌。他回忆在20岁之前根本不敢谈女朋友、去酒吧（同期的贝克汉姆、巴特等人女朋友已经换了很多），每天的时间都花在练球、休息上。依靠自己的勤奋，以及良好的职业素养，最终赢得了老爵爷[⊖]的认可，和其他天赋过人的球星一起上场比赛。

我认识一个男生。在高考那年的春节，客厅里亲戚在搓麻将，房子外鞭炮声此起彼伏，他坐在书桌前温习功课，亲戚说他太用功、太自觉了。工作以后去外地出差，他身边永远带着技术书，同屋的同事要看电视，没有地方可以看书，他便走进刚洗完澡的浴室，坐在潮湿的空气中看别人的代码，惹得同事说你也太认真了。现在，这个男生步入了男人的年纪，出了三本书，提交了近20项发明专利，为各类技术杂志写了50多篇文章，工作中也拿到了很多奖项。这个男生就是我，周明耀。高中时代曾经被物理老师认为脑子不太聪明、为人倒是挺厚道的我，依靠着自己的努力，逐渐在技术圈有了知名度，得到了技术人员的认可，谢谢诸位恩师、良友。

作为一名技术团队管理者，你要做的是比别人更早接触新技术，更深入地理解系统架构，全面提高管理团队所需的方方面面的能力，而所有这些都需要勤奋。在我熟悉的技术领域里，我能够快速地和人沟通、指导、给出自己的想法，这是日积月累的能力，而不是看几天书就能说出来的。正如马尔科姆·格拉德威尔（Malcolm Gladwell）在《异类》[⊖]中提出的一万小时定律：要成为某个领域的专家，需要一万个小时的训练。按比例计算就是：如果每天工作八个小时，一周工作五天，那么成为一个领域的专家至少需要五年。另一个学习的判断标准是：学习一个东西如果只是能听懂，还不算学会；如果能够按照所学去执行，也才学会了一半；如果能够把所学的东西用自己的语言表达出来，让别人听懂，这才算

⊖ 曼联前主教练弗格森，依靠自己培养的青训球员，带领曼联从普通球队成长为世界顶级球队，因此被英国女王授予爵士爵位。

⊖ Outliers:The Story of Success。

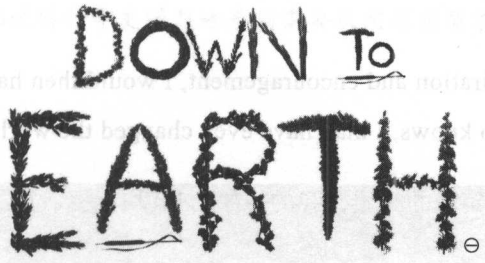
差不多学会了。

在软件行业，没有在专业上的持续学习和成长，是不会成功的。技术变革如此迅速，如果不坚持学习，可能大学毕业几年后就被淘汰了。在不断更新的软件开发行业中，这并不意味着你不需要使用大学里学习的知识，而是除此外还需要不断地增加对新的技术、工具、方法、框架的深度认识和理解。

5. 脚踏实地

“古往今来，能成就事业，对人类有作为的，无一不是脚踏实地攀登的结果。”

——钱三强



网上有这么一个未经考证的段子：“在伦敦闻名世界的威斯敏斯特大教堂地下室的墓碑林中，有一块名扬世界的墓碑。其实这只是一块很普通的墓碑，粗糙的花岗石质地，造型也很一般，同周围那些质地上乘、做工优良的亨利三世到乔治二世等二十多位英国前国王的墓碑，以及牛顿、达尔文、狄更斯等名人的墓碑比较起来，它显得微不足道，不值一提。它没有姓名，没有生卒年月，甚至上面连墓主的介绍文字也没有。但是，就是这样一块无名氏墓碑，却成为名扬全球的著名墓碑。每一个到过威斯特敏斯特大教堂的人，都被这块墓碑深深地震撼着，准确地说，他们被这块墓碑上的碑文深深地震撼着。”在这块墓碑上，刻着这样的一段话：

“ When I was young and free and my imagination had no limits, I dreamed of

⊖ Down to Earth= 脚踏实地。

changing the world.

As I grew older and wiser, I discovered the world would not change, so I shortened my sights somewhat and decided to change only my country. But it, too, seemed immovable.

As I grew into my twilight years, in one last desperate attempt, I settled for changing only my family, those closest to me, but alas, they would have none of it.

And now, as I lie on my deathbed, I suddenly realize:

If I had only changed myself first, then by example I would have changed my family.

From their inspiration and encouragement, I would then have been able to better my country, and who knows, I may have even changed the world."

译文是："当我年轻的时候，我的想象力从没有受到过限制，我梦想改变这个世界。

当我成熟以后，我发现我不能改变这个世界，我将目光缩短了些，决定只改变我的国家。

当我进入暮年后，我发现我不能改变我的国家，我的最后愿望仅仅是改变一下我的家庭。但是，这也不可能。

当我躺在床上，行将就木时，我突然意识到：

如果一开始我仅仅去改变我自己，然后作为一个榜样，我可能改变我的家庭；在家人的帮助和鼓励下，我可能为国家做一些事情。然后谁知道呢？我甚至可能改变这个世界。"

回到技术管理领域，很多技术团队管理者，无论是初次担任还是资深人士，都会出现看不清自己的情况，即说白了容易自大。人只要开始自大，就容易犯错

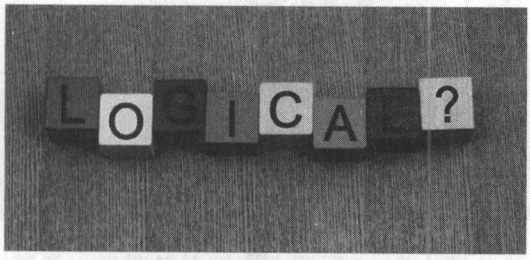
误：技术、产品上容易定位失误，管理上容易霸道待人。举个例子，如果现在我们决定做一个类似分布式数据库的存储组件，相信只有一小部分人觉得应该自己做这个组件的底层技术，大部分的人会选择使用开源框架实现，这其实都没有错，具体取决于你产品的定位、公司实力。我想说的是，你在做这些决定之前，有没有做足充分的技术调研、产品调研工作，能不能尽量详细地说清楚已有框架、产品的实现原理、优缺点、是否适用于你的产品场景等，这些都是前期技术调研的工作，也是体现我们脚踏实地做事的一个环节。

另外，请你记住这句话："如果不吸取失败的教训，那么失败就仅仅是失败。"

6. 逻辑能力

"没有一门科学比逻辑科学更强烈地感到需要从问题实质本身开始，而无须先行的反思。"

——黑格尔



程序员的工具是编程语言，日常活动和主要工作包括设计（Design）、建模（Model）、编码（Code）、调试（DeBug）、重构（Refactor）、沟通（Communicate）、学习（Learn）和思考（Think），所有这些工作对一个人的逻辑能力要求都很高。

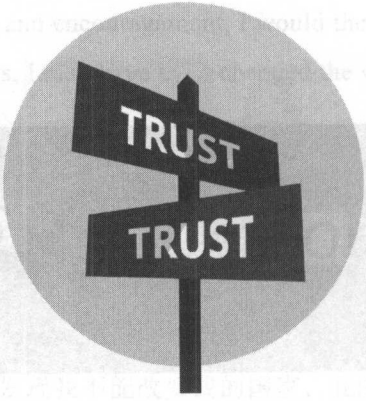
为什么有的程序员能够很快解决 Bug，有的则不能？首先，你需要对整个事情的全局情况有深入的了解。在发生问题、出现 Bug 时，你知道得越多，能想到的可能原因就越多，然后根据 Bug 的现象一一对应，排除不能解释症状的猜测。知识越渊博，你能一眼找到问题所在的概率就越大，如果知识存储不够，任你想破脑袋也无从下手，因为问题可能出在一个你不了解的地方。

每一次的技术调研、技术选型都是我的最爱，但有时候我也是非常地痛恨。这是因为，技术调研或者技术选型，通常是对一个陌生技术领域中技术、框架的初次研究，需要快速提炼出测试方案、测试用例，短时间内掌握调研内容，然后通过测试用例运算得出的数据进行分析，而这些数据的正确性就非常考验一个人的逻辑能力，稍有不注意，就有可能被下属忽悠了（也许他自己也把自己忽悠了）。一个人的逻辑能力属于软实力，它是决定你是否可以承担团队领导职责的关键因素。

7. 公平、透明

“举事以为人者，众助之；举事以自为者，众去之。”

——淮南子



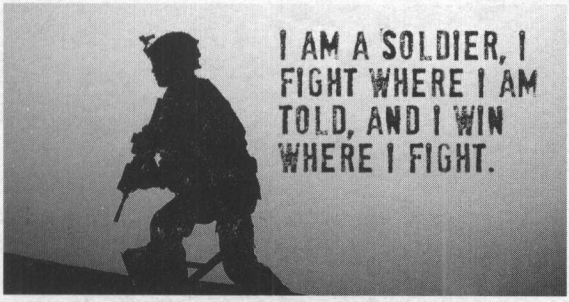
GE 创立于 1892 年，创始人是爱迪生，核心价值观是“诚信”。在 GE 进入中国时，发生了一起违背“诚信”的案例：一位员工多报销了 50 元业务招待费用。这件事 GE 是这样处理的：得知这一事情后，从美国派人前往中国调查此事，花费近 1 个月的时间，超过万元的差旅费，最终核实情况属实，做出解除与员工劳动合同的决定。GE 之所以这样做，一方面是让全体员工知道 GE 核心价值观的重要性，以及公司维护它的决心，另外一方面也是对员工负责，给员工一个公平公正的处理程序。因此，一方面对价值观要坚持，对违反价值观的行为要有相应措施；另一方面过程要公正、可复盘，这样才能做到让员工、让社会心服口服。

回到我们的团队管理主题。我们管理技术团队的风格，应该从十几年前的管理工人思维，逐渐提升为与知识分子沟通的方式。无论是绩效考核，还是职位升迁，亦或是问题处理，都不应该由少数几个领导通过闭门会议的形式完成，而应该是通过制定逻辑严谨的规章制度，通过类似任职资格考核（团队建设部分会深入介绍）的方式完成初步考察，然后通过事情责任制方式明确个人的成绩、责任，最后通过有员工代表、相应部门代表参与的会议讨论决定，这样可以让整个团队朝着健康的方向发展，而不会让一部分人感觉被边缘化，从而滋生小团体。

8. 一线作战精神

“平时的艰苦训练是战时胜利的保证，这才是对士兵的最大仁慈。我是一个很坏的家伙。我要让他们尝试每一分钟的地狱生活，然后我又为他们痛哭！”

——乔治·巴顿，美国陆军上将



两年前的一个夏天，我带领的团队购买了 10 多台服务器，用于分布式计算实验。收到到货的消息后（到货时间很难估算，所以没有提前联系货车），一般领导都会让下属或自己联系货车，但是我评估了一下时间，需要等 2 周时间。我们等不起，所以我带着 1 位下属，骑着自行车到了仓库，然后和大客车师傅（多厂区之间运送工作人员）商量是否可以把机器放在行李层运输，得到肯定的答复后我俩把服务器搬上车，然后我让他骑车回去，我跟着车一起回，回来的路上给其他组员打电话，全部到楼下集合，自己抬上去。一个下午完成服务器运送、搭建、调试网络等所有工作（路由器、插线板、网线等我已经提前准备好了）。这是我所理解的一线作战精神、作战方式。

我在 InfoQ 发表文章后，一位网友向我吐了很多苦水，他的大意是：“团队管理者请了长病假，CTO 直接接管这个团队，但是几个月了从来没有开过会，大家不知道要做什么”。针对这个问题，我的认知是，无论你是 CTO、CEO，还是 CFO，既然承担了团队负责人的实际工作，那么就应该每天召开例会，不断地抽时间和大家一起讨论架构、业务流程、技术难点、技术方向，这是你的职责，不要来谈你有多忙，你的职务有多高，如果真的很忙，你可以任命一个临时团队管理者替你承担责任，而不是让团队处于流浪状态。简而言之，这位 CTO，你失职了，不管你喜不喜欢，既然你直接管理团队，就要以身作则。

上面这张图里的银杏叶并不在杭州五星幼儿园内生长，是老师带领孩子们去学校外一点一点收集、搬回来的，对于小孩子来说，过程很辛苦，但是最终大家收获了一个美美的下午，拥有了一个永远的美好回忆。

9. 决策能力

“当今的时代，可以说是‘变化的时代’，重要的是，要具有把握这种变化的时机感和预见性。”

——彼得·德鲁克

朋友向我抱怨他的领导永远不会参与技术讨论，每次开会也不会提供任何技术建议。我反问他，在任何的技术层面，特别是需要对架构、技术选择进行决策时，这个人能不能给你意见或者帮助？他说不能。我觉得我这位朋友可以换个部门了。

管理人才的决策能力主要由这样几个方面构成：

1）**开放的提炼能力。**开放的提炼能力是指管理人才以开放的态度，准确和迅速地提炼出解决问题的各种方案的能力。包括两个基本要素：第一、要以开放和包容的思想及态度尽可能广泛地获取决策方案，特别是不要局限于传统的解决办法之中，要善于“借外脑”来帮助判定决策方案；第二、需要对各种决策方案进行提炼，以把握各种方案的本质和核心，正确地评估每个方案的条件及效果，分析各个方案实施的可能性。

2）**准确的预测能力。**决策与预测是密不可分的，要具备卓越的决策能力，首先应具备准确的预测能力。预测是决策的基础，决策是预测的延续，预测的目的是为企业的决策提供准确的资料、信息和数据，在正确预测的基础上，选择符合企业发展的满意方案。

3）**准确的决断能力。**即能从众多的决策方案中选取满意方案的能力，以及危机时刻或紧要关头有当机立断的决断能力。这种能力是进行科学决策的关键能

力，误选、漏选会使企业造成重大损失或与成功失之交臂。对此，必须把握以下几个主要标准。一是所选方案实施的条件要具备。若条件不具备，则要弄清获得该条件的代价是什么。二是所选方案要与企业的宗旨和决策目标相符，若不符则不可取。三是所选方案要能被决策方案的受益人及相关利益人所接受。四是所选方案要能被决策方案的执行者所接受。好的决策方案只有执行和实施后才能达到最终的目的，因此要注意决策方案的可接受性。五是能正确评估决策方案的风险。有些人在选取决策方案时只看到“乐观”的一面，而没有考虑环境的可能变化，这种“乐观”情绪往往会给企业造成重大损失。

决策总是需要我们在不同意见的基础上做出判断，而不会是一致性意见的产物。快速的决策通常错误率较高。决策的目标应该是明确的，决策的结论应该是简单而清晰的，而不是建立在一大堆假设的、复杂的数学推断上，这也是为什么阿里在进行双 11 的负载能力需求统计时，最终选择采用全链路压测[⊖]，而不是像早前通过 Excel 表格对服务资源消耗进行大致计算。

培养决策能力应注意以下几点：

1）**克服从众心理。**从众心理是指个体对社会的认识和态度常常受到群体对社会的认识和态度的影响。从众行为者的意识深处考虑的是自己的行为能否为大众所接受，追寻的是一种安全感。从众行为者认为群体的规范、他人的行为是正确的时候，就会表现出遵从；当他认为群体的规范、他人的行为并不合适，而自己又没有勇气反抗时，就会被动地表现为依从。从众心理重的人容易接受暗示，他们依赖性强，无主见，人云亦云，容易迷信权威和名人，常说违心的话，办违心的事。决策能力强的人，能摆脱从众心理的束缚，做到思想解放、冲破世俗，不拘常规、大胆探索，因此他们能独具慧眼，发现一般人不能发现的问题，捕捉到更多的成才机遇。

2）**增强自信心。**拥有自信心是具有决策能力者明显的心理特征。没有自信

⊖ 通过接入线上服务器，采用模拟双 11 超高流量方式测试整个业务流程上所有模块的负载能力，这种方案要求模拟场景真实，且不能因为模拟造成线上应用奔溃。

就没有决策。增强自信心首先要有迎难而上的胆量。温斯顿·邱吉尔就说过："一个人绝对不可在遇到危险的威胁时，背过身去试图逃避。若是这样做，只会使危险加倍。但是如果立刻面对它毫不退缩，危险便会减半。决不要逃避任何事物，决不。"其次要变被动思维为积极思维。"凡事预则立，不预则废。"平时善动脑筋，关键时自然敢作决定。再次要培养自己的责任感和义务感，跳出个人的小天地，如此你的自信才能坚实可靠。另外平时交往注意要选择那些有自信心、敢作敢为的人，时间长了，看得多了，你必然会受到积极的影响。

3）**决策勿求十全十美，注意把握大局**。做事务求十全十美，不想有任何挫折或失误，那只能作茧自缚。如能识大体，把握大局，权衡出利弊得失，当机立断，才能尽快达到自己理想目标。持之以恒，你的决策能力和水平就会很快提高。

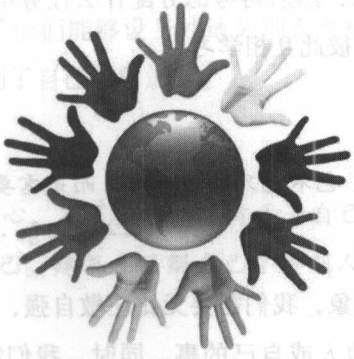
团队领导者需要具备决策能力，决策能力又是与自身的技术积累，以及吸收新技术、新知识的能力密切相关的，保持一颗开放、积极的心态，保持对技术的严谨调研态度，以数据说话，这样才会做出正确的决策，而不是盲目决策。

此外，成为领袖的人，都是从小事做起的，不要好高骛远。

10. 开放姿态

我可能会不赞成你的看法，但是我尊重并捍卫你说话的权利。

——伏尔泰



曾经在微信上看到某位投资者的文章，他曾经对一家技术公司进行评估，各项指标都比较满意，但最终放弃的原因是：该公司 CTO 的管理方式非常野蛮，他不允许别人对他提出的技术方案提意见，提出意见就会被他冷嘲热讽，最终团队中全是老实巴交、毫无追求的程序员。基于 CTO（首席技术官）这样的管理方式，这个公司的技术能力和积累，无论当下还是未来，发展都会受到限制。

我们需要倡导“开放、共享、追求极致”的团队文化。人才是我们最大的财富，所以要建设以人为本的团队文化，创造出沟通顺畅、敢于挑战、喜欢创新的团队氛围。在团队文化建设方面，主要倡导的是互联网文化、极客精神。有的工程师，为了开发一个高效的算法，通宵熬夜，结果虽然只提升了短短的 10 毫秒，但这就是追求极致的精神。试想在一个亿万级调用量的场景下，提升 10 毫秒，效果将是质的变化。据说国外某家投资银行为了提升 1 秒的系统研发投入几千万美元，可见其价值。

“罗马军团是人类历史上最具机动性的部队。他们由 8 个人组成一个最小单位的战斗组，而 8 刚好是当时一个帐篷可以容纳的人数”。这是 Twitter 的高级副总裁 Chris Fry 在谈到稳定的团队建设时所提到的例子。大多数科技公司（包括 Twitter）刚起步时，每个员工都会被分配到一些工作，但这都是被动进行的，项目来了，成员才会被快速地集结起来，这让作为管理者的你形成了一个在短期里快速分配人手的习惯。在公司发展早期，这或许是好的，但从长远来看，则需要改变这一习惯。作为一个管理者，你的重点应该首先在于创建出有良好化学反应的、能把事情做好的团队，然后再考虑分配什么任务给他们。好的团队，不仅可以很好地协同工作，还能彼此互相学习。

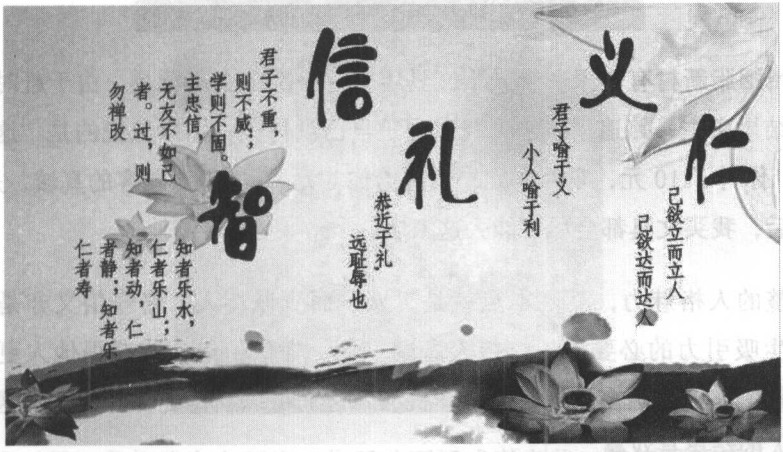
11. 为人处世

“你不能只言如其言，也不能只行如其行，而是需要言行一致”

——塞西尔·威廉姆斯，著名牧师

为了公司、自身的形象，我们需要无比庄敬自强，公平待人，不可欺负弱势的人，也不可以做损及他人或自己的事。同时，我们需要一个谦卑的团队负责

人，谦卑的人不会固执己见，而是会虚怀若谷地聆听他人的言论。伟大的人物也不可能整天仰望山巅，他也会蹲下来为他的弟兄濯足。快乐与金钱和物质的丰盛并无必然关系。一个温馨的家、简单的衣着、健康的饮食，就是乐之所在。漫无止境地追求奢华，远不如俭朴生活带给你幸福和快乐。内心能够保持这样心态的团队领导者，在面对很多困难时，因为有家人的支持，有团队的支持，有内心平和心态的支持，会更能克服困难，勇往直前。

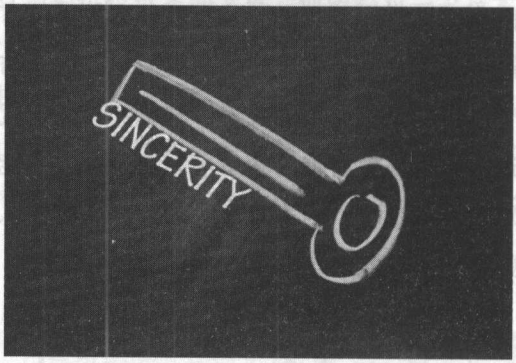


有一次我去成都参加校招面试，由于面试是在酒店进行的，需要面试人员在房间外等候，有些学生就横七竖八地坐在、躺在 HR 的床上，可是真的有这么累吗？这不是在自己家，这样的行为既不礼貌，也不卫生，晚上让别人怎么睡觉，这是礼仪，是做人的基本道理。我尊重那些站在 HR 门外的同学，那样既体现了他们良好的家教，也体现了他们能够设身处地为别人着想，而不是以自我为中心的良好素养。他们已经拿到了自己的附加分。

12. 真诚

“有了真诚，才会有虚心，有了虚心，才肯丢开自己去了解别人，也才能放下虚伪的自尊心去了解自己。建筑在了解自己了解别人上面的爱，才不是盲目的爱。”

——傅雷



杭州胭脂新村有一家“晨光”文具店。有一次我去买玩具，由于赶时间买完就走，结果店主一路追了出来，我以为是自己没付钱，结果他说的是“这是你的6元钱，你付了10元，东西4元，需要给你6元。”这是对顾客的真诚，是诚信。从此以后，我买文具都会尽可能去这家店。

完整的人格魅力，其基本点就是真诚，而真诚待人，恪守信义亦是赢得人心、产生吸引力的必要前提。待人真诚一点，守信一点，能获得他人更多的依赖、理解，能得到更多的支持、合作机会，进而可以获得更多的成功机遇。友善真诚待人的结果是双赢。

在工作时，我们和团队成员是上下级关系，也是合作关系，共同运营着这个团队，我们需要真诚待人，有什么问题可以在会议上或者一对一谈话时指出，说出自己对他工作的看法。生活中，我们是朋友，有什么事情都可以一起讨论，想办法解决。我相信每一个人都喜欢真诚的上级，我也不例外。其实现代工作环境，特别是科技行业的工作环境，非常开放，大家在一起的工作时间一般来说不会超过5年，既然时间这么短，就应该互相坦诚，一起就事论事，正面指出缺点，才能互相帮助，共同进步。

记住，你若待人真诚，别人也会用心对你。

13. 宽容

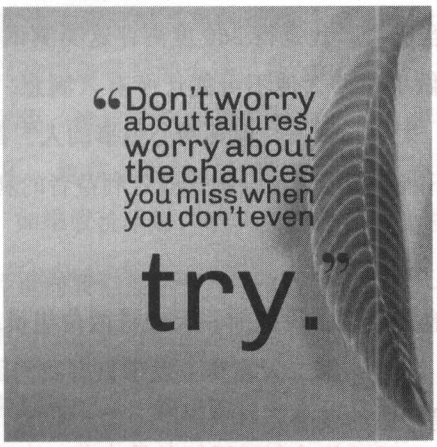
“我每看运动会时，常常这样想，优胜者固然可敬，但那些虽然落后而仍跑至

终点的竞技者，和见了这样竞技者而肃然不笑的看客，乃至是中国将来的脊梁。”

——鲁迅

“庆祝你的成功，在失败中寻找幽默。别太把自己当回事儿，你放松，你周围的人才能放松。享受乐趣，并始终保持激情。”

——山姆·沃尔顿，沃尔玛创始人



很多时候，我们确实会面临工作无功而返或者辛苦一年都碌碌无为的情况，但这并不是我们不努力，更可能是产品或者技术的风口还没到，我们走得有点过快了，或者方向有点走偏了。另外一种情况可能是团队成员的个人能力问题，有时候确实会遇到一些思维比较慢的同事，你布置了调研任务，也花费了时间做了详细解释，但是他就是理解错了。这种情况下我觉得需要看这位同事的以往工作绩效，以及他对这项任务的工作态度来做出反应。我常说的是，这个人如果自身能力较弱，不愿承认，那就是作死。如果自身能力很强，也很积极做事，这个时候我们应该多放权，让他自己思考，他也会乐于如此。如果自身能力不强，但是他对自身弱点有认识，工作态度很好，这个时候犯错了，我们不能过度责骂，应该宽容对待，先问清楚原因，然后和他一起分析过程，帮助他成长，只要他成长了，我们的团队整体也会受益。

我们团队有一个 90 后的小伙子，平时不怎么爱说话，但是当工作中遇到问

题时，别人可能尽量躲避，他却会主动要求去研究解决。有一次，在系统发布时Zabbix老是报警说已经下线的应用不可用，非常烦人。在例会中我问谁有兴趣研究一下，虽然这事和他一点关系都没有，但是他主动承担了这个任务，只花了半天时间，就找到了问题所在：有一个接口，只要将节点的状态置为维护状态，就不会在发布时做无效报警了。这件事虽然不大，但却能体现他做事的主动性。

另外一点，管理者也要能包容下属的缺点。很多公司喜欢用360度考核法[⊖]来评测一个员工的综合能力，我觉得360度测评这类测试不太适用于科技公司，因为如果一个人想要做事，那么他不可能让所有人满意，如果让所有人都满意了，那就是城府较深，太用心做人，但凡用心做事的人，都会有机会得罪人。科技意味着不断革新，不断革新就可能会损害既定利益者的利益。

14. 仔细

有一次部门去外地搞活动，我提前一天安排两位组员去超市买了很多零食、水果，也预备了午饭。结果，第二天在车上发现只有我们团队有东西吃，其他团队都未提前准备。这是仔细的表现。

我在生活中也是一个很仔细的人，也喜欢仔细的人。仔细的人一般做事都比较负责，愿意承担责任，也懂得抓细节。抓细节是作为一名技术团队管理者必须做到的。所谓的管理浮于表面，一般都是说管理者不关注细节，例如不参与设计、不参与代码审核，而只是高喊要注意开发设计、注意开发质量，这样的领导对于技术团队来说，尤其是一线技术团队经理来说，是不合格的，也是不能服众的。

我觉得一个仔细的人，他一定也是一个善于观察的人，而善于观察、思考其实也是一个团队领导者所必须具备的品质，否则他的团队一定会在某一阶段或者一直处于混乱状态。

⊖ 360度考核法是常见的绩效考核方法之一，其特点是评价维度多元化（通常是4或4个以上），适用于对中层以上的人员进行考核，EMBA、MBA等常见经管教育均对360度考核法的实施流程及组织要求有所介绍。

15. 终身学习

“一个人只要去行动，去坚持，去积累，就会在翻山越岭后看到胜利的曙光，会享受到成果以复利方式增长，会收获时间的玫瑰。”

——沃伦·巴菲特

《Becoming Warren Buffett》（成为沃伦·巴菲特），这是 HBO2017 年拍摄的关于巴菲特的最新纪录片。在这部纪录片里，巴菲特坦然褪去自己身上众多的光环，展露了自己最真实的一面。影片里的大部分镜头，都献给了巴菲特的家人，用于回忆他的过去，以及记录平常的生活。比如巴菲特每天早上上班，开车路过麦当劳时会去买一份早餐，然后带到办公室享用。他的桌子上也随时摆放着一杯他钟爱一生的可口可乐。他在上班路上打趣道：“如果今天公司股票价格涨了，我就买 4 块 2 的套餐；如果股价不好，哈，那我就买 3 块 8 的。”

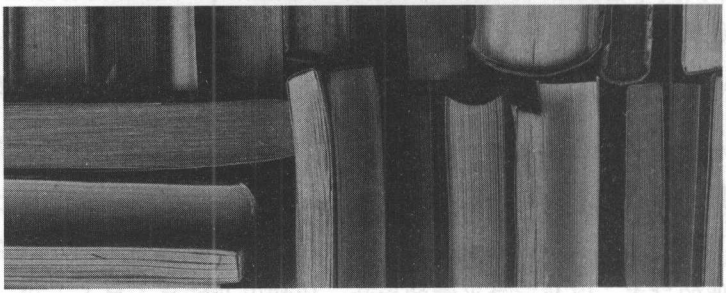
同时，这部纪录片也向我们展示了一个事实：一个人一生如果想要获得过人的成就，就注定与读书和终生学习形影不离。

巴菲特每天绝大多数的时光，都是独自一人在自己的书房或者办公室静静地度过。他每天会按时起床，花大量的时间阅读各种新闻、财报和书籍。他的办公室没有电脑，没有智能手机，只有身后书架上的书籍和一桌子摊开的新闻报纸。他每天就坐在那里阅读和学习。时光静静流逝，六十年如一日。终生阅读和学习的巴菲特，即使在 84 岁的高龄，还掌管着全世界最大的投资公司，保持着敏锐的大脑和思维，以及对工作和生活的热爱。

功利性的学习是非常狭隘的，收获也是非常有限的，但是终生学习的回报却是不可估量的。爱因斯坦曾经说过，复利（Compounded Interest）是这个世界上的第八大奇迹，那些理解并学会使用它的人将最终获得巨大的财富。那些不理解它的人会付出巨大代价。复利的效果不仅可以应用在财富的积累，也能应用在知识的积累上。

当你在时间的长河中坚持不懈地学习，你的知识将会在复利的作用下持续地累积和增长，最终的收获和回报会远远超出你的想象，而实现终生学习的最佳途

径，就是阅读大量优秀的书籍。



读书不仅是获取知识的手段，更能够培养出优秀的人格，这是甚至比获取知识更要有意义的一件事。我们也许没有办法决定我们的出身和阶层，但是我们每个人都可以通过读书和终生学习，为自己塑造一个优秀的人格，实现个人的价值提升和阶级突破。

回到技术管理领域。端午节和同学聚会，有位在国内某著名电商公司就职的同学提到他们公司有一位 P8 的研发团队管理者辞职了。原因是领导认为他的技术有些落后，换了一位 90 后当主管，这让他感觉被架空了。于是大家开始围绕 90 后进行讨论，认为管理 90 后太难，因为他们太有个性，也是他们现在的职业威胁。我放下了手上的餐具，发表了自己的看法："我认为，这和 90 后没有关系，再过几年，你会发现 00 后也开始成为你的威胁。无论国内某通信大厂的 34 岁清理传闻，还是刚才你说的这个情况，实质上是这位管理者跟不上团队的节奏了。对于老板来说，下属需要出成绩，既然管理者不能承担岗位职责，唯有让能够担责的人上来，这实质上和年龄无关。"这些话听起来很残酷，但是事实真的是这样，试想，某通信大厂怎么会轻易让有十多年经验的优秀人才离开？在需要重用人才的时候，离开的大多是无法被重用或是薪资和能力严重不匹配的人。我们谁也没有办法阻止公司的变化，无论是主动离开，还是被动离开，我们都无能为力。除了希望政府从劳动法层面上对我们给予保护以外，唯有保持自己的学习能力，你才能以变应变[⊖]。

⊖ 事实上，我认为当下已经极少有一位技术工作者可以在一家企业干一辈子了。

记住，一个人只有严于律己，长年累月专注做好一件事情，并且坚持终生读书、学习，才能享有并保持随之而来的成功、荣誉和认可。

16. 时间管理

“从不浪费时间的人，没有工夫抱怨时间不够。”

——杰斐逊

我们普遍存在着这样一个认知误区：多花时间 = 态度好 = 产出高。然而即便是在泰勒管理理论的时代，这都是被否定的结论。

GTD（Getting Things Done）是一种高效的管理时间的方式。个人感受就是要划分任务，重要的事情优先完成，用一定的时间专注处理一些事情，然后再休息，再专注，周而复始，这和番茄工作法比较类似。通常，最好的方式是在每一天早晨花上一定的时间来规划一天的安排。

作为一名管理者，很多时候自己所能支配的时间大多有限，只有学会集中精力、善用碎片时间，才能让工作效率更高。如果管理者在多个行业或项目中拥有较多的资源，那么需要自己设定优先次序，而且遵循要事第一的原则，坚守其设定的优先次序。战线如果太长，势必造成精力无法集中，反而会影响一些原本优先级较高的项目的执行。

记住，你最重要的资产就是时间，对时间要吝啬，但同时对那些真正需要它

的人或事要慷慨给予。找到方法来管理好时间，你的生活将会过得更好。比如，你可以把手表调快几分钟，这样每次会议你都会有几分钟的缓冲时间，确保准时到达会场。

17. 以人为本

20 世纪 80 年代的克莱斯勒 CEO Lee Iacocca，在其畅销自传中是这样介绍他的管理哲学的："人、产品、利润。人是第一位的。" Iacocca 先生的意思是，应当以人为本。如果你有卓越的人才，并给予相应的尊重和鼓舞，让他们能够积极参与工作，那么你会做出卓越的产品。如果你有卓越的人才和卓越的产品，利润就会随之而来。作为管理者，你的工作中很重要的一项；绝对不可忽视你团队的人，并始终要坚持以人为本。

我们其实在很多场合都能感受到这一点的重要性。如果团队员工认可你的管理，那么他们会在项目危难之际坚定地跟你站在一起，支持你，陪伴你一起解决一个又一个难题。我们都知道，只要坚持，总会有胜利的一天。

18. 保持身体健康

"钱能买到一切有价的东西，独独换不回健康，也唯有生命才是无价的。"我们经常会听说有人得癌症了，有人又猝死了，这些例子数不胜数。

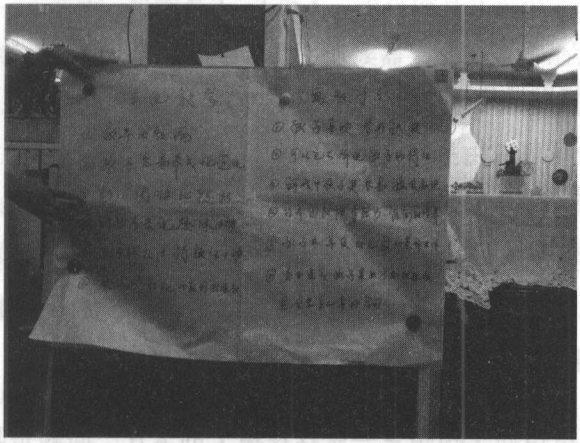
虽然我能理解健康的重要性，但经常无可奈何。一个现场问题出现了，我们要一直忙到凌晨，可能偶尔还需要通宵应对。但是空下来以后，一定记得好好睡一觉，把睡眠补回来。因为只有保持身体健康，才能更好地带领团队。

1.3 带领技术团队心得

1.3.1 幼儿园家长会感悟——传统型教育 VS 创新型教育

2016 年 9 月 23 日，女儿上中班以后的第一次家长会。

老师给家长出了一道题目：“论述传统授课型幼儿园教育方式 VS 新型游戏型幼儿园教育方式”。家长们临时组队，分为 3 组，大家都踊跃发言，讨论过程很是激烈，正方、反方、中间派观点均很明确。不得不说，现在的家长思维都很活跃，都很有自己的看法和见地，而且讨论中很尊重其他人，很不错。



其实，传统教育方式目标比较明确，就好比我们带领一个技术团队。通过传统教育方式培养出来的优等生，适合带领目标比较明确的团队，比如告诉你这个团队需要在指定时间内完成哪些需求点，最好是系统架构、数据流走向都已进行限定，这样就比较容易完成了。

游戏型幼儿园教育方式培养出来的优等生，创新能力十足，很有主见，能够在很短时间内找到自己的定位，懂得如何快速地与人打交道，找到适合自己的团队成员，一起抢夺有限的游戏资源，让自己和团队能够最大程度地玩耍。但是可能他们不太喜欢按部就班地工作，在那样的环境下容易使他们产生抵触心理。

从幼儿园的培养方式和孩子们的分类来看，可分为创造性和遵从性两类，也就是需要针对这两类孩子采取不同的培养方式。我认为，这两种方式本身没有好与坏，关键在于你把它用在哪里。例如传统的外包行业、传统行业的 IT 部门，采用传统教育方式培养出来的学生就能够较好地完成目标，因为他们习惯了惯性思维，能够理解、包容，在不需要自己设定目标、设计创新型产品的时候，更能

适应行业所限定的一切。而对于游戏型幼儿园来说，它是为不确定具体目标的新型行业、企业培养人才，因为在那些领域，你无论带领一个团队，还是负责一个产品的设计，没有人会告诉你明确的目标，没有人会告诉你这个团队的技术方向是什么，没有人会告诉你做一个什么样的产品可以获得收益，全都要靠自己，只有在最短时间内能够找到自己方向的管理者才能胜任这样的岗位要求。所以，是走技术管理路线，还是走技术专家路线，可能从幼儿园就决定了，多关注一下自己的孩子，对于他的成长，真的很重要。

1.3.2 自主循环

女儿就读的五星幼儿园是杭州市的老牌重点幼儿园，也是杭州市幼儿教育的示范改革试点单位，它尝试采用新型教育方式培养孩子的自主能力、动手能力、社会适应能力，隐式地让孩子的智商、情商同步发展。

举个例子，伊伊同学很想拿一张挂在树上的卡片，问题是除了那张卡片以外，还有其他很多张卡片，明明可以轻松地拿到其他的卡片，但是她一定要那张自己够不着的卡片。她看着柴老师，但是柴老师故意假装没看到她，这样僵持了一会，伊伊去找其他老师帮忙拿到了卡片。这个故事说明伊伊是一个很有自己主见的孩子，想好了就一定要，并且不会轻易改变想法。另一方面，也说明了她善于寻求外部帮助，而不是一个人苦干。

女儿个性比较内向，胆子比较小，对于新事物不太敢参与。有一次，她看到小朋友都在爬一个只有入口，没有明确出口的袋子，她感觉很恐惧，连续在周围晃了好一会儿，就是不敢尝试。这时候灵灵出现了，她很热情地邀请女儿一起爬过去，女儿还是不敢，灵灵就亲身示范。看到灵灵的尝试，女儿也找了一个袋子开始爬，但是由于看不到出口，她爬爬停停。灵灵爬出来以后立即跑到女儿这边，拉起她的袋子，大声说：“你看，我们能够看到光”。女儿很受鼓舞，很快就爬出了袋子。这个例子说明内向型的孩子不太敢于尝试新鲜事物，但是如果有人帮助，他们也是可以突破内心的恐惧，完成别人可以完成的工作。

有一次周末我遇到其他团队的一些同事，他们赶过来加班，但是我没有看到他们的领导。问了才知道，因为项目紧急，几个实际做事的人组织了一个小会议，讨论了当前的问题，发现需要周末来加班，不然赶不上预定的测试时间，所以他们自己决定周末过来加班，这就是自主循环。

我们带领一支团队，可能是 1、2 个人，也可能是几十个人，甚至上百人，我们不可能做到事无巨细、一一谈话，很多时候需要分小组讨论，也要依赖于个人自主工作态度，这就是我们所说的自主循环。比如说布置一个技术调研工作，你讲清楚了测试方案、测试用例，接下去的工作必然是由工程师分头行动，如果没有一个很好的自我驱动，互相讨论、检查规则，即没有形成个人和小范围内的自我行动循环，如果每一行代码都需要你来帮他编写，每一个外部工具都需要你去搜索引擎搜索，那基本上你只能同时干一件事情了。

1.3.3　技术人员心态成长四个阶段

皮亚杰（瑞士）将儿童和青少年的认知发展划分为四个阶段：感知运动阶段、前运算阶段、具体运算阶段和形式运算阶段。他认为所有的儿童都会依次经历这四个阶段，新的心智能力的出现是每个新阶段到来的标志，而这些新的心智能力使得人们能够以更复杂的方式来理解世界。虽然不同的儿童以不同的发展速度经历这几个阶段，但是都不可能跳过某一个发展阶段。同一个个体或许能同时进行不同阶段的活动，这明显表现在从一个阶段进入到一个新阶段的转折时期。

技术人员同样分为这几个阶段，在初级阶段，他们需要在你的帮助下感知需求、架构，逐渐拥有自己设计简单模块的能力，再接下来他们又有了复杂模块的设计能力，逐渐通过实战训练，最终能够独立承担大项目的管理和设计工作。总结起来大致可分为这几个阶段：

- **感知阶段**：这个阶段技术人员刚刚工作，有很多知识要学，有很多经验需要积累。这个阶段的技术人员，内心其实存在矛盾，一方面想要坚持技术道路，一方面又对未来存在迷茫，迷茫可能是因为工作压力，也可能是对

于工作和生活的本质矛盾。有一次开高管见面会，一位刚工作的女生向女性副总裁提问，“我觉得对当前的工作内容挺满意的，也想未来成为技术大拿，但是有一点我不喜欢，我不喜欢下班后的生活依然和工作相关，我看到很多同事下班后都在加班，我希望他们不加班，和我一样准点下班回去享受生活，这个怎么才能办到？”对于这类问题，其实很难回答，副总裁回答得很棒：“没有哪一个人的成功是可以很轻松得到的，我到现在也经常通宵，不是因为我不喜欢睡觉，而是因为我对工作有追求，既然有追求，你就要学会付出。”其实这个女生的心态在这个阶段很正常，因为这个阶段的人还没有完全成熟，也许将来他们会离开技术领域，也许他们会沉淀下来，专心搞技术，这都无所谓，关键是要想清楚，多和前辈聊聊，交换思想，知道怎么做才能达到自己的目标，并付诸实施，如果你觉得自己做不到，那说明你选错岗位了。

- **深入阶段**：这个阶段的技术人员已经拥有了一定的技术能力和学习能力，也开始清楚自己的未来发展方向，如果到了这个阶段能留下来的人，应该是喜欢技术的，未来一段时间也会继续从事技术工作。我们对于这类人的指导，应该从实践着手，给他们一些独立思考的机会，给予他们实际的指导，就事论事，就架构论架构，就代码论代码。
- **突破阶段**：这个阶段的技术人员已经遇到了一定的发展瓶颈，他们渴望突破瓶颈。如果你不能很好地解答他们当前的问题，他们很有可能会流失。这个阶段，你需要让他们有完整的独立思考机会，安排一些从头开始的工作给他们，然后积极地与他们进行讨论，适度授权，让他们一边自我实现，一边听到你的看法。
- **创新阶段**：这个阶段的技术人员已经不是你能够给予指导的了，他们是在按照自己的思路进行创新，你要做的是给予掌声、认真倾听、给予意见，最好你能一同参与创新活动，这样你们才能一起进步、创新。

1.3.4 找到属于自己的圈子

在成为管理者，特别是成为具有一定管理经验的技术管理者之后，我们需要

能够发现并组织一个属于自己知识交流的圈子，这个圈子需要定期碰面，讨论各自正在从事的工作以及软件开发中遇到的具有挑战性的问题。这种方式会帮助彼此从对方那里获得帮助，时不时还能共同总结出一条至理名言或经验法则，然后各自可以把这些至理名言和经验法则整合到管理方法中，并与团队分享。

在这类圈子里，我们可以有机会与团队管理者、总监、首席技术官交流，眼界也会更加开阔。可以从自身的行业经验出发，体会别人的经验，而不只是根据书本上看到的经验法则。我们还可以分享自己开发或使用的工具，以及在创业和各种规模的公司里工作所获得的经验。

大家知道，学习有几种方式。第一种方式是通过网上的碎片信息来学习；第二种方式是通过读书来系统地深入学习；第三种方式是通过和别人交流来学习，这种方式可以增加与优秀的人的交流，毕竟“听君一席话，胜读十年书”的道理是正确的。我们自己在工作或者创业的过程中，很多问题都是无法在网上和书本上找到答案的，公司里可能也没有适合的人一起交流讨论，这时候圈子就发挥了作用，和行业里的专家不断交流、碰撞，从而快速成长。

这里我以自己作为示例。我觉得圈子分为工作内和工作外两部分：

工作内：可以尝试多参加各类技术论坛，通过这些论坛你可以了解新的技术点，认识其他公司的技术人员，一群有共同目标的人很容易凑在一起，然后可以组建社交群，时不时地聊聊技术，或者碰碰面。除了论坛之外，你可以多上上国外的知名技术问答网站，对你绝对有帮助。你还可以思考自己是否可以参与其中，我在 InfoQ 和 IBM 开发者论坛都开设有专栏，不是我闲暇时间多，而是我要给自己不断输出技术成果的目标和压力，进而不断提升自己的技术能力。

工作外：除了工作以外，你可以提前几年为自己未来的发展方向布局，比如你的目标是成为 CTO，那你要明白，当前你的朋友圈可能没有这样职位的人，或者没有可以让你走上这个位置上的人，那么你需要找机会去结交这样的人，或者是那些大企业的领导，也许，将来哪一天目标就实现了呢？目标总是要有的。你可以选择加入一些高管俱乐部，或者知识分子组织，比如九三学社。

1.3.5 何时成为技术管理者

一位来自浙大的师弟看到我写的那篇文章《这是你希望的 Leader 吗？[⊖]》后加了我的微信。他说自己工作 3 年，现在带领 6 个人的团队在做后端技术开发，但是觉得自己还处在技术上升期，想去阿里这样的公司锻炼，但又有点放不下现在的岗位 title，所以来咨询我的看法。我建议他先去锻炼，无论是不是去阿里（也不要太相信大公司，还是要以所在部门承担的技术工作职责作为优先考虑条件），都需要加深自己的技术深度。倒不是说技术管理我能做，他就不能做，不是这样的，我的认知是技术积累是一辈子的事情，只有到了一定程度，即你可以花较少的时间学会一项新技术或者架构的时候，你才能去做技术管理的工作，因为技术管理工作会占用你很大一部分工作时间。

我记得华为某产品总监这样评价李一男，他回忆了一个场景：李一男完全不懂某一个产品技术，但在去会议室的路上听取了产品的简单介绍后，他就能够在会议中无障碍地参与这次技术讨论。我不知道这件事情是否真实，但我觉得这是有可能的。因为技术本身是相似的，比如分区概念，可以用在 GC 的设计上，也可以用在 HBase 的设计上，在 Oracle 里也有这个概念，而且设计思路和目标基本一致。那么当一个人的技术基础很好时，他就会很容易理解你所讲的技术，能够在最短时间内掌握，这就是一个人的学习能力，有点类似于慕容复[⊖]的“以彼之道还施彼身”，只有功力深厚的人才能做到。再者，你怎么知道他没有学过？也许私下他早就看过很多资料了，杰出的程序员都是有强烈的主动学习动力和能力的。

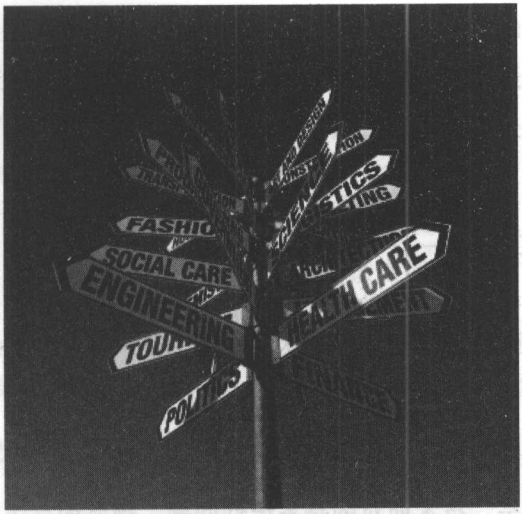
回到我们的主题，什么时候成为技术管理者？这没有恒定的标准，但是我的建议是，先打好技术基础。当你觉得每天只有 50% 时间可以做技术的时候，但你还是能够紧跟技术，甚至比你的大多数下属都了解细节的时候，你就可以尝试给自己加上技术管理的工作量了。

⊖ 原题是《我所理解的技术管理 1》，新媒体编辑觉得太朴实了，改了题目。

⊖ 金庸作品《天龙八部》中的人物，出身于武林世家姑苏慕容。其真实身份是于五胡十六国时期入侵中原，并建立多个“燕国”的鲜卑族贵族慕容氏余脉。

1.4　个人职业发展

1.4.1　对于职业的选择



二十几岁刚毕业的时候，选择公司的因素会有很多，但是我个人认为兴趣应该是最主要的。校招面试的时候，我也更喜欢那些有明确技术兴趣和方向、知道自己想要做什么的候选人。如果做一件不喜欢的事情，那么很难有动力在今后的工作中不断前进。

说到钱，现在的毕业生都很关心钱和待遇的问题，但是目前同等水平职位，待遇差距不是很大，市场相对透明，在差距为 1000 ～ 2000 元的情况下，钱不应该是一个主要因素。因为如果发展得顺利，是很容易和同时期入职的同学在收入上拉开差距的，这绝对不是 2000 元的水平。所以要想发展得好，还是要靠建立在兴趣上的个人努力，以及另一个关键因素；领导或者导师。这对于一个刚入行的新人来讲是尤为重要的，所有的职场价值观形成可能都与此相关。

随着职业生涯阅历的增长，对人的评价大多都会从技术能力高低转到价值观是否合拍上来。尽量做一个简单而靠谱的人，但是想做到确实需要一直持续努力。在不同的人生阶段，经过不同的阅历，每个人的选择都会不一样。有的人已

经在大公司打拼过，见过了大公司是如何运作的，有了更广阔的视野后，就会想要尝试下自己是否能亲自创造出一些东西。

IT 行业是个知识更新非常快的行业，当然其他行业也在发展进步，但是相对而言还没有这么强的节奏。同时，IT 行业的机会很多，每次失败最好都能从自己身上总结出原因，这样才能继续向前，从而提升自己。选择很重要，但先要想清楚自己想要什么，选择之后，被放弃的选择就不再重要了，重要的是怎么把选择的事情做好。

回到技术管理工作。一直有同仁问我，什么时候可以转为技术管理？如果在军队里面，一个士兵说我杀敌本领不行，是不是可以升为将军？同样的道理，最好是技术能力比较强之后再转管理，水到渠成，技术不行的人即使转了管理，也难使人信服。

从技术转管理本身就是一个很大的转变，一旦跳进去，就很难爬出来了。我认为最大的差异就是，技术面对的是系统性问题，需要的是逻辑思维能力，是智商，而管理需要面对的很多是非逻辑思维能力，是情商。系统问题需要智商，而人是活的，你要理解人，就需要情商，情商是什么？我个人理解，情商就是如何站在别人的角度看问题。

最后对职业的选择做一些个人的理解陈述，我觉得，只有自己喜欢的工作才能真正地全身心投入。我们都很幸运，生活在一个不需要太担忧有上顿没有下顿的时代，不需要一家十几个人挤在一间房里，我们完全可以摆脱父母那辈的很多束缚，让自己的兴趣做主，做自己喜欢的事情，爱自己喜欢的人，至于结果，要看你的付出，也等时间来证明吧。

1.4.2 工程师的等级

前苏联著名物理学家 Lev Davidovich Landau 提出过一个衡量物理学家水平的郎道等级。他把世界上的物理学家分为了五级，即第一等物理学家的贡献是第二等的十倍，第二等是第三等的十倍，依此类推。其实，在各个行业里，不同层次

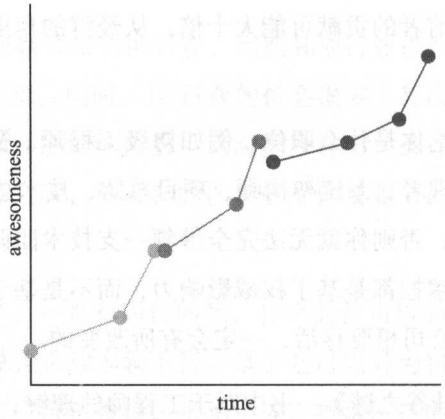
人才的贡献也都大多如此。比如在 IT 行业，乔布斯认为他的合伙人沃茨尼亚克就抵得上 20 个二流的工程师。从成本上看，一流工程师的收入可能是二流工程师的两三倍，但是，前者的贡献可能大十倍，从经济的角度来讲，采用最优秀的人才是最合算的。

我的认知是，无论你是什么职位，例如初级工程师、资深工程师、项目团队管理者、技术团队管理者、系统架构师、项目总监、技术总监、CTO 等，首先一定要具有工程师特性，否则你就无法完全带领一支技术团队，无法给他们技术上的支撑。因为所有的掌控都是基于权威影响力，而不是基于职位的，即便反过来了，也是暂时现象，公司想要存活，一定会有所改变的。

引用吴军博士《硅谷之谜》一书中对于工程师的理解，我也认同这样的说法，即工程师按能力可以分为五等，大家应该努力让自己向上一等级靠拢：

- 第五等工程师，能够独立设计和实现一项功能的人。这是对工程师的基本要求。
- 第四等工程师需要有点产品头脑，也就是说他们在做一件事之前，要知道所做出来的东西是否有用、易用，是否便于维护，是否性能稳定，等等。除了要具备产品设计方面的基本知识，还要具有一定的领导才能，能从头到尾负责一个产品的生命周期。这在很多硅谷的公司里是一个高级工程师所应有的基本素质。
- 第三等工程师，可以做出行业里最好的产品。他们与第四等工程师有着质的差别，这不仅反映在技术水平、对市场的了解、对用户新的了解以及组织能力等诸方面，而且也反映在悟性的差异上。
- 第二等工程师是那些可以给世界带来惊喜的人，他们在其工作的原创性以及对世界的影响力上和后面三等有巨大差别，例如扎克伯格。
- 第一等工程师是开创一个全新行业的人，这些工程师不仅在技术和产品等各个方面与第二等工程师有了质的差别，而且在经验和管理上也是好手，他们通常也是企业家，并通过自己的产品改变了世界。这一类人常常是可遇而不可求的，例如冯·诺依曼。

1.4.3 不要计较当下职位



我清晰地记得我的一位同事和我说："只要想，谁都可以当上 CTO！这么多机会，你们又都是聪明人，最终不过是选择不同规模的公司做上去罢了。"

最近有两件事对我震动很大。一件是发生在朋友所在的公司。他们的市场总监算是公司的元老，在过去近十年时间里与公司共进退，把生意从几千万做到了十几个亿，功德圆满，受人尊敬。美国总部也对她青睐有加，不断褒奖。然而从去年开始，受宏观经济形势影响，中国市场生意开始下滑，苦苦奋战了一年多之后，公司总部宣布任命新的中国管理层，这位市场总监，就被赶走了。年届 45 的她，不知道还能不能找到一份理想的工作。

另一件事则直接发生在我另一个朋友身上。她是我见过最聪明的人之一，美国名校毕业，顶尖咨询公司的工作经历，后来成为了一家 500 强企业的管理层。用她自己的话说，"位高权重"。她情商又非常高，在公司里可谓左右逢源。我一直觉得她是我朋友中成功的典范。但事情反转得很快：公司原来的 CEO 退休了，换了一个和她不对味的新老板，公司模式调整，她管的那一块业务突然无足轻重了。她跟我抱怨说："每天都被挤兑，做的事情又是那么的没有意义！"也想要跳槽，可是外面适合她的工作并不多。所以在被解雇之前，她可能要在这样不愉快的环境里煎熬很久。

虽然上面两件事不同于我以往的认知，但在企业里面成功，不可否认仍依仗两个方面：**要么你聪明，会交际，懂政治，被人认可；要么你做出成绩，贡献巨大，被认可**。我觉得，技术人员还是选后面一条比较合适，持续地积累自己的技术，最终你的成长会由量变达到质变。

1.4.4　坚持做正确的技术管理

最好的领导，思路敏捷而清晰，做事务实而高效，永远没有废话，永远雷厉风行。和他共事，绝对不会轻松。他不会讲段子哄你干活，他也不会笑眯眯地安抚你的情绪。每周的例会，是一场例行考试，每个人都小心翼翼、如履薄冰。如果顺利，他会扔个梗让大家笑一笑，然后收起笑脸继续挑战你；如果不顺利，他会打断你，生生把你拍在墙上，毫不留情，但有理有据。从此，你再也不敢造次。他不会掩藏不悦，他对愚蠢的忍耐度很低，他说变脸就变脸。如果被他认定为“不求上进”，你在他眼中就成了废物，从此不会给你好脸色。有时候，你会觉得，他简直是个刻薄冷酷的资本家，是个不近人情的吸血鬼。但事实的真相是：评判一个管理者的好坏，从来不是看测验的民意，而是看输出的成绩。良好的干群关系和群众基础，有助于达到目的，但却不是最终目标。一个好的经理人，就是要做出好成绩，对老板负责；一个好的 CEO，就是要做高利润或股价，对股东负责。这才是职业的素养，这才是商业原本的逻辑。

格雷格·波波维奇是马刺的主帅，他追求细节，甚至有些偏执。在训练场中，他不满足于现状，总是不断提出严苛的要求，一视同仁，不会因为球员身份不同而区别对待。而在平常的生活中，他充分发挥自己的幽默天分，和弟子之间彼此信任、亲如一家。有人说波波维奇是“所能遇到的内心深处最善良的人”。在邓肯的退役发布会上，这样一位铁血教头声音颤抖、几度哽咽：“想和他告别不可能”。场面令人动容。

李嘉诚教育子孙，做人的最高境界是“仁慈的狮子”。仁慈是本性，但单靠仁慈，还无法成功。要有狮子的力量，才能赚钱养家，才能保护亲人，才能反抗欺压。“做个好人”和“做个好的管理者”，其实并不冲突。

世上最讨厌的词是“烂好人”。善良发展到极致，就变成无原则的妥协。

对于什么是正确的技术管理，其实没有正确答案，也不会有这样的标准答案。但是对于我们的工作，是可以有衡量标准的。你有没有给予下属工作内容上的帮助，例如技术难题解答、沟通能力提升、项目管理能力传授等，每一条都可以细分出很多，总而言之，只有你的上级、你的下级，他们才能评判你是否是在做正确的技术管理。让别人满意，不容易。**不断地调整自己的工作方式，不断地加强自己各方面的能力，特别是改正自己的弱点，这些才是你应该去关注和为之努力的。**

1.4.5 抛弃通道思维，建立雷达视角

我们很多企业都有专门的考核方式，一般对于工程师来说，分为专业通道和管理通道。对于技术管理者来说，很多时候两个通道都需要被考核。

很多人会认为，只需要按照通道的要求进行积累就可以了，也确实是这样，如果你真能完全满足通道的要求，你的能力绝对是很强的。但是事情往往不是这样的，以技术管理者为例，他除了技术能力通道以外，为什么还需要走管理通道？因为没有管理，整个团队会乱，技术与管理结合的方法论掌握不好，整个团队的绩效输出都会差，你别指望纯技术通道的人能够输出业务部门能够理解的产物。也就是说，需要能够抛弃单一通道的约束，尽量做到全面的能力建设，即抛弃通道思维。

但从整个职业生涯来看，我们很难做到一帆风顺。换一个领导，或者公司出现了严重的问题，都会让我们的“通道”出现危机，而这时候，最重要的是能力，而不是公司所给予你在“通道”上的认定资质。如果你的能力不再是“职业通道”里的生存技巧，而是基于自身的天赋发展出的技能，那么你不会再沿着一个狭窄的方向前进，而是以自身为圆心，等距离向外的探索，好像雷达扫描一样。当你发现一个机会落入你的雷达区时，意味着你的技能与之是匹配的，那你就可以努力抓住它。可能会有很多工作机会落在你的雷达里，你可以选择其中有趣的

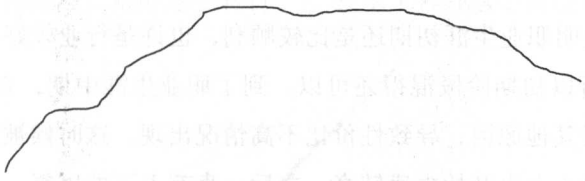
进行发展。虽然像传统职业通道里的工作一样，机会也会此消彼长，但是当你不再只看一个方向，而是有一个 360 度的视野时，你的选择会变得很多。你的技能越强，你的雷达探索的面积就会越大。在这样的世界观里，我们不再追求人人都去做团队管理者，而是追求做一个靠天赋、靠能力吃饭的人。转变的过程是痛苦的，跳出原有模式是需要极大勇气的，但最后的收益很诱人，即自由，更强的安全感，还有乐趣。

我们这个行业的人，不容易。我们需要坚持学习，不然随着年龄的增加，你的优势会逐渐消失。抛弃通道思维，建立起自己学习新知识的雷达视角，建立起自己的朋友圈，扩大雷达视角范围，对你一定会有用的。

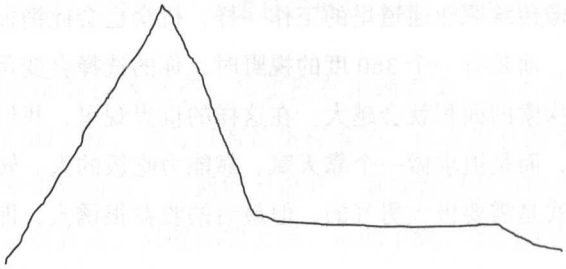
1.4.6　收入浮动曲线

IT 从业者在整个职业生涯的收入趋势，这个话题很多我的读者开小窗问我，我做一个自己的想法说明。

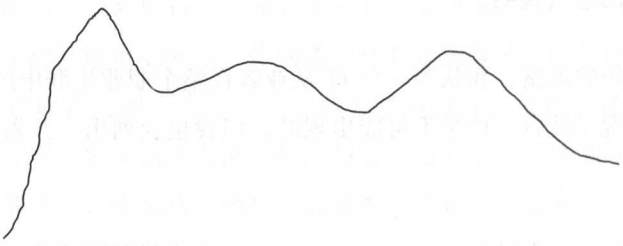
根据我实际的观察，我认为一位 IT 从业者在整个职业生涯中的收入趋势可能由以下七种情况（最后三种是不可能出现的，但我也会列出，并做解释），我逐一说明。



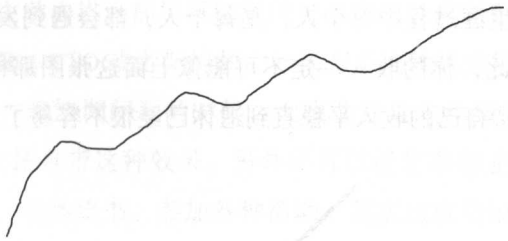
这条曲线说明从业者前期个人工作比较努力，因此收入一直在不断增长（所处行业也需要比较景气，以下同），进入职业生涯中期（比如 35 ～ 45 岁之间），收入开始趋于平稳，职业生涯末期由于职务逐渐远离中层、高管范围，所以收入开始小幅度下滑。虽然退休前几年开始收入有所下滑，但总的来说职业生涯较为平稳，这主要还是依靠个人的不断努力。



这条曲线说明职业生涯初期，由于个人会忽悠、行业也是爆发性增长，双重前提下收入会不断增长，但是由于自身能力较差，进入职业生涯中期后，由于一些变故（行业不景气、公司倒闭）等等原因，被裁员后找不到行业内的工作，不得不转行到其他行业，导致收入剧降。这类人其实当下是存在的，只不过是因为行业爆发性增长，他们混入了 IT 技术领域，但是终究是会被揪出来淘汰的。IT 企业想要一直保持生存，必须要有不断的技术创新，因此必须依赖强有力的技术人员，没有技术能力的人、不爱学习的人终会被淘汰。



这条曲线说明职业生涯初期还是比较顺利，也许是行业较好、个人也还算有点技术功底，所以初期阶段混得还可以。到了职业生涯中期，自身出现了瓶颈，可能是性格或者其他原因，导致性价比不高情况出现，这时候被动地接受调整岗位或者离职，收入上也开始由盛转衰，之后一直有上下的反复，但是始终没有能够回到巅峰，职业生涯末期出现大幅度下滑。工程师很多都是性格内向，在你到了一定的年纪时，如果没有特别突出的技术能力、自身也没有其他能力的话，势必会成为企业的鸡肋，也就有了裁员的说法。这种做法其实不是裁员，而是对于组织技术活力的调整，没有哪个公司愿意主动调整工作能力强、工作责任心强的又拥有 10 年、20 年丰富工作经验的员工。



这条曲线说明从业者整个职业生涯一直在不断地学习、在不断地增强自己的综合能力、不断地自我积极调整以适应企业需求。这类人很成功，能够保持大体上的收入不断增长，一直到退休功成身退。能够真正做到这样的技术人员很少，大家共勉吧。

有一次去武汉出差途中，出租车司机和我聊起了他的儿子，28 岁，做 IT 工程师，月收入 1.3 万，跳槽的要求是不能出差、不能加班、双休，因此，他的收入定格在了 1.3 万，而他表哥在杭州某电商大厂工作，已经达到 2.6 万。司机师傅形容他儿子希望一辈子保持一个还可以的中等收入，不求发展，也不想为了工作累着自己，就像上面这条横线。很可惜，现在是市场经济社会，不是过去那种一个厂可以让你混一辈子的时代了，所以上面这条曲线是不可能出现的情况。没有哪家老板会愿意让你在他这里混着，或者愿意从别的企业来接盘你这样的人。有这类想法的人很危险，最好尽早转行，IT 技术工程师这个岗位不适合你。

有一位读者和我讨论问题，询问 35 岁左右年收入多少合适，我报了一个平均的数值，他觉得少了，因为他 28 岁已经快接近这个数值了。他的想法是像上面这张图一样，一直在增长，没有一点波澜，这不太现实 IT 行业从业者，大家

心态要放好，职业生涯过程中每个人，是每个人，都会遇到发展瓶颈，甚至是失败，这很正常。因此，你的收入一定不可能像上面这张图那样保持持续性增长，人到中年，能够保持自己的收入平稳直到退休已经很不容易了。

这种曲线趋势出现概率较低，一毕业收入达到顶峰，然后持续下降，应该不会有谁是这样的情况，真有的话，好好静心看点书学习吧，你会改变现状的。

1.4.7 CTO 角色解释

CTO 角色是很多工程师梦寐以求的，除了这个角色之外，技术 VP 也是不错的定位。这里我们对这两个角色的差别进行解释。

CTO 角色

CTO 是公司第一号技术大师，他对于技术非常敏感，需要具备对于技术发展的深远见解，保持公司在技术上的竞争力。CTO 需要保持自己对于专业领域内前沿技术的不断搜寻，也要对可能影响公司的技术方向的旁系领域进行调研。

CTO 需要非常热爱技术，并且愿意自己动手尝试新的技术，一般来说 CTO 也会有自己的“CTO 办公室”，有几位全职或者兼职的下属，他们会帮助 CTO 一起做技术调研或者原型产品。

也正是由于 CTO 常年耕耘于技术前沿，所以他也承担了公司 blog 的重责，应该是博客文章的重要贡献者。

国内外的技术领导风格其实有很大的不同，在美国，每个公司的技术副总裁、CTO 和高级架构师都很注意影响力，我们经常看到，当公司内部有技术分享

的时候，有很多人主动去讲，都尽可能展现自己在技术或者管理方面的长项。一个工程师、技术主管、CTO 或者架构师，如果有了这种技术领导力，当他跟同事一起讨论问题或者一起协调问题的时候，大家往往会主动倾向于他，“他说的事情一定不会假”，往往自带这种效果。另外还可以经常参加业界的分享，做各种技术委员会的委员，包括出书、参加各种活动。其实这也是给自己的职业发展铺一条路，所以塑造影响力不仅仅为了企业，也有利于个人职业发展。

美国 Amazon 的 CTO Werner Vogels 博士是一位很具有代表性的 CTO，正如前文所说，他拥有自己的技术博客，我对他最近的帖子进行了截图：

30.06.2017: Improving Customer Service with Amazon Connect and Amazon Lex
28.06.2017: Stop waiting for perfection and learn from your mistakes
21.06.2017: Amazon DynamoDB Accelerator (DAX): Speed Up DynamoDB Response Times from Milliseconds to Microseconds without Application Rewrite.
21.06.2017: Expanding the Cloud – An AWS Region is coming to Hong Kong
07.06.2017: Unlocking the Value of Device Data with AWS Greengrass.
19.05.2017: Weekend Reading: Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases.
03.05.2017: Faster, higher, stronger: How the digitalization of industry is redefining value creation
12.04.2017: Coming to STATION F: The first Mentor's Office powered by AWS!
07.04.2017: Back-to-Basics Weekend Reading: Twenty years of functional MRI: The science and the stories.
04.04.2017: Välkommen till Stockholm – An AWS Region is coming to the Nordics
30.03.2017: Back-to-Basics Weekend Reading: An Implementation of a Log-Structured File System
24.03.2017: Back-to-Basics Weekend Reading: Deep learning in neural networks
15.03.2017: Amazon Makes it Free for Developers to Build and Host Most Alexa Skills Using AWS
13.03.2017: How companies can become magnets for digital talent
10.03.2017: Back-to-Basics Weekend Reading: The Foundations of Blockchain
04.03.2017: Back-to-Basics Weekend Reading: Why Do Computers Stop and What Can Be Done About It?
24.02.2017: Back-to-Basics Weekend Reading: Byzantine Generals
10.02.2017: Back-to-Basic Weekend Reading: Monte-Carlo Methods
03.02.2017: Back-to-Basics Weekend Reading - Bloom Filters

大家可以看到，博士除了写一些和所在公司相关的深入技术、产品设计分享以外，他也会针对一些基础知识进行分享，于是就有了“Back-to-Basics”系列。我上个月开始写的技术杂谈系列，也是源于他的启发。

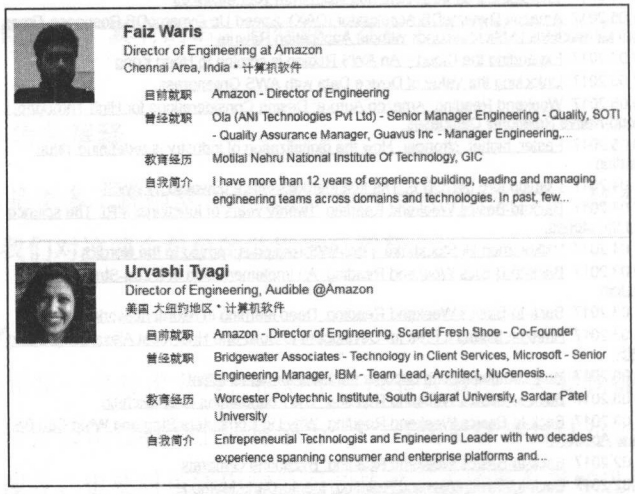
一些公司设有首席架构师，根据我对多家美国大型科技企业的研究，首席架构师一般是 CTO 的备选人物，这是因为首席架构师和 CTO 关注的领域较为相近。首席架构师和 CTO 之间通常的差距是经验，首席架构师有点类似于小号的 CTO，积累足够的经验后，他们就可以走向 CTO 岗位。

技术 VP

简单地说，技术 VP 的职责是交付软件解决方案，确保业务健康，只有业务健康，工程师们的努力才有价值，团队才有可能继续发展。

前面提到了 Amazon 的 CTO，我也在 Google 上搜索了 Amazon 公司的技术 VP，但只在 LinkedIn 上搜索到他们的资料，按照名字去搜索，无法找到博客或者发表过的文章。

搜索了亚马逊的技术 VP 或对应职位，最后在 LinkedIn 里找到了一些，但是他们都没有开设个人技术博客。



技术 VP 需要是一位杰出的管理者、团队构建者，其工作内容涉及招聘、沟通、问题解决等。技术 VP 的工作是确保工程团队内的每一个人成功，即他的工作是解决成功过程中遇到的问题。

CTO 和技术 VP，两者应该是合力开发产品，而不是谁领导谁的关系。

创业型公司容易出现这种情况，由于没有第一时间找到或者招聘到一位靠谱的技术 VP，导致开发过程总是被各种情况打断，团队的氛围很差，工程师和产品经理之间隔阂很深，这些都是因为没有一位称职的技术 VP 在产品开发流程内

进行统筹管理。

对于一家科技公司来说，CEO 和技术 VP 之间的合作至关重要。CEO 负责明确需求，将市场的需求精简为产品需求，技术 VP 则从专业角度上确保团队可以按时发布产品，即便在 CEO 时不时地修改需求的情况下，优秀的技术 VP 依然可以确保按时完成任务，并且他也起到了 CEO 与技术团队之间的润滑剂作用，将压力挡在了技术团队门外。

技术 VP 通常包括以下职能：

1）人员管理：对于少于 10 人的团队，技术 VP 直接管理团队内所有员工，对于 10-100 人的团队，技术 VP 下设多位经理，他负责管理这几位经理，对于大于 100 人的团队，公司会上设工程总经理，他会向这位总经理汇报。

2）项目管理和执行：技术 VP 需要对产品的开发、交付负责，他需要在公司内部糅合各个团队之间的关系，包括开发部门、资源部门、审计部门等，确保产品开发的正常进行。

3）财务管理：这里指的是工程部门范围内的财务管理。具体包括人员投入、原型设计费用、设备费用、出差、娱乐等。

4）技术领导：首先他需要和 CTO 一起制定公司的技术战略规划，然后根据规划制定技术演进到产品的 RoadMap，确保产品上的技术能够保持创新。对于架构师这个职责，技术 VP 可以自己担任，也可以委派组织内的其他人担任。

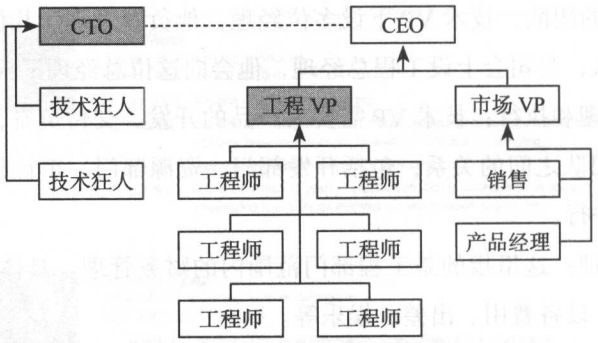
5）战略发展：技术 VP 不一定是技术大牛，但是他一定是跨学科人才，因为他的工作需要和多个不同部门的人打交道，包括市场 VP、业务发展 VP、制造部门 VP、运维 VP、CEO、CTO、COO，共同开发公司的战略和产品战略。

我也搜索到了一位比较典型的技术 VP，Coursera 的 Richard Wong。Richard 出生于香港，11 岁时第一次接触计算机程序设计，斯坦福硕士毕业之后，在微软干了 11 年分布式软件开发工程师，而后又在 LinkedIn 呆了 4 年，接着就去了创业公司 Coursera。他认为杰出的程序员和一般的程序员的巨大差距在于需要多久你可以把自己的想法转变为代码。

同样的问题也会落在技术 VP 身上，要不要写代码？从我个人的理解，研发人员很重要的是需要一位能知道他们、了解他们的大哥。你也要参与代码过程，目的在于可以通过写代码去理解大家、了解大家在想些什么。你作为技术领导者，必须融入这个氛围，了解一线员工他们在做什么，作为他们的发言人，代表技术团队跟公司管理层争取一些利益和福利，大家会觉得这是我们自己的带路人。

总结

对于 CTO 和技术 VP 的工作 CTO 和技术 VP 的工作是有明显差别的。我们可以用下面这张图来表示：



需要明确的是，公司业务稳定之后的 CTO 和工程 VP，这两个角色所承担的责任是存在很大区别的，两个角色都对业务的扩展有重要作用。个人感觉，CTO 与工程 VP 之间的关系是紧密的，工作上各有各的分工，CTO 规划技术愿景，工程 VP 更多负责业务愿景。

第 2 章

团队建设、人员管理

尽管有规范的软件工程流程（如 CMMI1-5 级），但是这些流程的发挥的作用其实不大。很多程序员开发软件并没有遵循这样的规范。即便遵循，也只是对流程有所改进，却无法将程序设计转变为纯粹的工程实践。此外，规范化的框架只解决了编写软件的流程问题，而没有涉及程序员管理的问题。遵循流程对管理程序员只能起到最低限度的帮助。技术团队管理者们仍然只能依靠自己的方法和工具来对程序员进行管理。

我们要懂得，欲管理人，必先了解人。程序员之所以难以管理，在于他们有着各种不同的个性。

本章主要介绍和解决以下问题：

- 什么是管理。
- 怎样组建团队？
- 如何管理团队？
- 影响团队管理的因素有哪些？

2.1 管理基础

管理人员是与人打交道，其任务是使员工能够协同工作、扬长避短。

——彼得·德鲁克[⊖]

2.1.1 什么是管理

首先聊聊什么是科学，如何定义科学？

科学通常需要按照前人的理论和数据继续向前走，深入钻研，管理靠的是个人实践。我不建议在没有准备好的情况下匆忙地转到管理上，你以为进入管理岗就可以学管理，你是拿你的短板和别人的长板比较，一定会遇到很多麻烦。观察、假设、校验，是一切科学实践的基础，管理活动也是类似的，你观察到某位员工最近好像不怎么说话了，产出也没有之前那么高了，就会假设他是不是遇到问题了或者想要跳槽了，最后通过直接沟通或侧面了解检验假设的正确与否。通过不断地刻意训练，慢慢积累自己的管理经验，再想着怎么去拿捏平衡，完成类似艺术形态的进化。

2.1.2 开发岗位解释

```
while(alive)
{
    eat( );
    sleep( );
    code( );
}
```

程序设计作为一种严肃的职业已经存在 70 多年了。全球目前至少有数百万程序员，这还不包括在校的学生、业余编程爱好者（他们非常认真地编写程序，但不以此为谋生之道）。1968 年开始，人们开始将程序设计这门艺术称为软件工

⊖ 彼得·德鲁克，生于 1909 年，被誉为“现代管理学之父”。它创造了知识型员工的概念，并预测了当今信息社会的兴起及社会对终身学习的需求。

程，从零开始编写新程序更像是在一张白纸上写小说。任何人都可以成为程序员，不需要接受正式教育，也没有必需的证书或考试。事实上，我认为只有特定类型的人才能成为程序员，只有非常特别的一类人才能成为杰出的程序员。从技术能力或分工来看，又分为应用程序员、系统程序员、系统架构师 / 工程师、开发团队管理者。

（1）应用程序员

应用程序员开发的程序通常给终端用户直接使用。程序包括文字处理软件、OA 系统、Web 浏览器、安卓应用程序等。一些应用程序员能够跳出代码本身的束缚，与应用程序的用户产生交互，真正从用户的角度看问题，从而很好地把握各种可视化、交互式设计之间的细微差别。如果让这样一位有天分的应用程序员与一名 UI 设计师合作，将产生一加一大于二的效果。

（2）系统程序员

系统程序员理解系统中所有组件的工作原理，包括客户端 / 服务端的操作系统和通信系统。以某个产品为例，系统程序员负责编写与硬件交互的设备启动程序，创建能够为设备驱动程序和应用程序提供运行时环境的操作系统，为其他程序员创建编译器和调试工具，还需要为其他程序员提供工具和服务以用于交付程序。

（3）系统架构师 / 工程师

大多数系统架构师 / 工程师都是从系统程序员做起的。在所有开发类职位中，系统架构师 / 工程师是最需要技术和经验的。要想理解所有相关的系统组件（操作系统、通信系统、数据库、在线 / 离线访问方式、安全性、硬件等）之间的复杂关系，你需要对所有这些技术和系统都有丰富的专业知识和实际工作经验。杰出的系统架构师 / 工程师可以让团队中的其他人表现得更好，他们设计的系统工作起来会更可靠，代码模块、系统架构也会更简洁。Google 公司的联合创始人 Larry Page 和 Sergey Brin 就是这样的人，他们在设计和实现上培育的优雅风格帮助 Google 公司在技术和商业领域都取得了伟大的成功。

（4）开发团队管理者

与项目团队管理者不同（项目团队管理者需要在自己规划的职业道路上花大

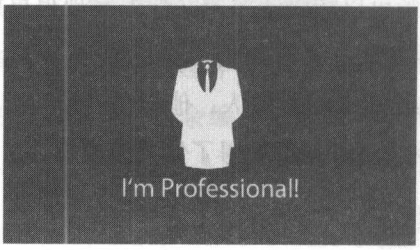
量的时间学习并获得证书），开发团队管理者一般都是优秀程序员出身，积累了较强的技术和架构能力，对业务知识也较熟悉，且拥有较强的人际关系处理能力，智商、经验、情商的三者综合水平都较高。这个岗位也是本书的重点目标人群。

2.1.3 团队成员品质

团队是为了实现某一目标而由相互协作的个体所组成的正式群体，是由员工和管理层组成的一个共同体，它合理利用每一个成员的知识和技能协同工作，从而解决问题，达到共同的目标。团队的构成要素总结为目标、人、定位、权限、计划。团队和群体有一些根本性的区别，群体可以向团队过渡。一般可根据团队存在的目的和拥有自主权的大小将团队分为三种类型，即问题解决型团队、自我管理型团队、多功能型团队。

组建一支团队的基础是需要雇佣合适的成员。一支整体作战能力较强的开发团队，它雇佣的员工需要具备以下几点素质：

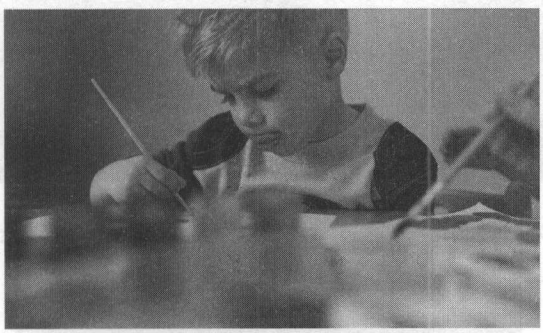
1. 职业素养



社交媒体上我多次收到这样的邀约："我们对您的技术背景和行业背景很感兴趣，我们是一家专业的咨询公司，受雇于国内 ×× 基金公司，正在针对您所处的行业进行技术、产品和竞争力分析，我们想对您进行采访。采访将采用电话访谈方式，耗时 40 分钟至 1 小时，我们会给予您 1000 元人民币的报酬。"这不是技术访谈，它涉嫌商业间谍，如果你真的照做了，你能不能收到钱我不知道，你的职业素养是要被打问号的。你以为钱是这么容易给你的吗？他们只会问一些不痛不痒的通识性技术问题？很大可能要的是你所在公司的产品发展情况、技术使用情况，最终以数据形式卖给一些公司或机构。

不要贪图小利，一个人的职业素养非常重要，让别人觉得你的职业素养有问题，你的职业生涯也会很危险。上面这种案例，我的回复是："这类事情请用公对公形式，您可以与我们公司市场部或者战略规划部对接，如果对私人进行采访，我认为这是一种商业间谍行为，对于您今天的联络，我会向公司报备。"

2. 做事专注



专注的人，更容易获得成功。爱因斯坦说过："一个人只有以他全部的力量和精神致力于某一事业时，才能成为一个真正的大师。因此，只有全力以赴才能精通。"

假期我带女儿去少年宫画画，老师要求在一个白色的存钱罐上面涂上颜色，这需要充分保持自己的注意力，女儿不是那种注意力很集中的人，事实上，我觉得没有谁是天生就注意力很集中的，孩子的认知、行为方式来自家庭的影响。我

对女儿反复说："无论别人在你周围谈论什么话题，或者他们发出什么声音，都和你无关，你现在要做的是专注于画画。"

同样的事例发生在《妙手仁心》这部电视剧，急诊室的主任在有人行凶的情况下，仍旧保持高水准的急救，这也是专注工作的充分体现。

回到技术领域，一名真正的技术牛人，他在做事的时候绝对是全神贯注的，否则他不可能在短时间内读懂别人的代码、找到问题原因、完成框架设计，这是技术专家的必备能力。

3. 乐于挑战

知乎上有这么一个问题："在 BD 工作，遇到的最大挑战是什么？"该问题回复很多，我列举一些，大家自己评判。

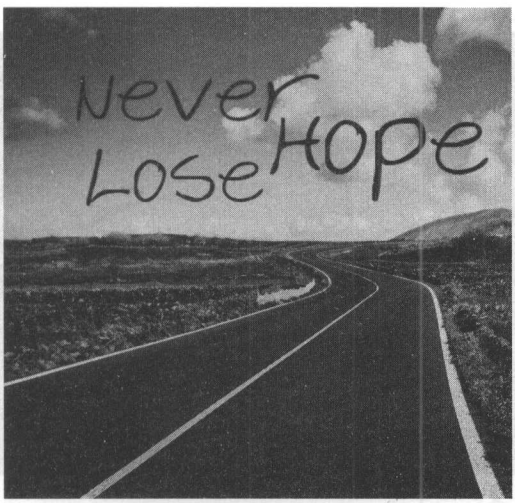
- "觉得最大的挑战是公司人多需求多，你实际的工作和当时的设想完全不同，这种情况下如何调整职业发展方向，或者是努力纠正，让自己回到原来设定的发展方向上来。"
- "盲目地扩张导致入职人员的素质水平跌得很厉害；越来越多的新人缺乏敬业精神；有能力的老人离职率很高；大多数 PM 缺乏专业知识；缺乏实在的指导人体系；上头什么都想做；大多数都做不出成绩；上班的时候山寨感强烈。"

- “各种会议和杂事儿一堆，能耗费你很多的时间和精力，不能专注到技术与架构本身。经理过分强调手下的工程师对项目整体、产品、预期的理解，有时候真觉得自己在产品上花的时间相当于半个 PM 了。”

在工作中你一定会遇到各种各样的挑战，你所需要做的是积极地迎接、应对这些挑战，而不是每次都退缩，越退缩就越容易失去机会，失去让自己过上自己喜欢的生活的机会。

当你加入一家新公司时，要挑一个比较棘手的难题（其他人尽量回避的）来解决。这样可以帮你快速积累经验，并赢得成为一名卓有成效的开发者所必需的信誉和尊重。我不太喜欢向别人要技术调研报告，事实上我最喜欢做的事情就是对一项未知技术做技术调研和预研，这会让我能够有机会挑战新技术，有机会让自己的视野更加开阔。

4. 永不气馁

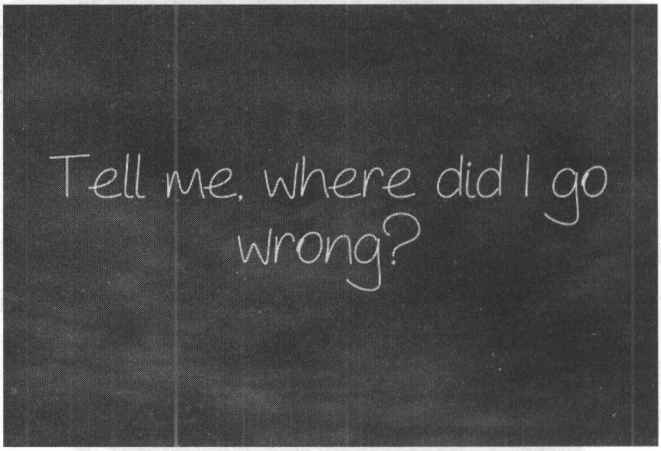


我在职业生涯中投篮失败 9000 余次，输掉了 300 场比赛，有 26 场比赛，我被委以投出致胜球的重任，却没能命中。我不断地遭遇失败，而这恰恰是我取得成功的原因。

——迈克尔·乔丹

2016 年我获得了公司的一个奖项，HR 让我写几句获奖感言，以便发表在公司刊物上，我写的是“永远不要放弃”，这也是我时常对自己说的话。这句话是原浙江省特级化学老师、杭州市高级中学退休老师郑克良老师对我说的。那时候我正在读高复班，对于未来我没有太清晰的想法，其实在那之前，我一直都是一个没有太多个人想法、内向、老实的孩子，我只是想通过高考让自己有一条清晰的路。遇到郑老师是在公交车上，他和我聊着天，了解了我的特殊情况（我读大学前的情况和一般城市里的孩子确实有些不一样，应该是我内心的那种坚持，主导着我不愿意进入真实水平对应的学校，一直想着从小的梦想——清华大学和美国哥伦比亚大学）。一次月考后，我拿着 30 多分的试卷去请教郑老师，他看着我说：“永远不要放弃了”，我开心地点点头。其实郑老师不知道，我一直都在进步，从第一次考试得 5 分开始，之后每次考试都能进步 10 分，这段经历也培养了我的自信，不懂的东西我学学就会了，没什么大不了的。

5. 承认错误



不管你做出多大努力，这个问题是不可避免的：在生命的某个时候，你会犯错。

错误难以消化，所以我们有时会孤注一掷，或回避他们，而不是直面它们。这时候，我们对事物的认知会产生偏差，导致我们需要寻求证据来证明自己所坚

持的信念。

心理学家称之为认知失调，即当我们持有两种相互冲突的想法、信念、观点或态度时所感受到的压力。《错不在我》的作者 Carol Tavris 认为："认知失调是我们在自我认知（我是聪明、善良的，我坚信这是真的）受到事实挑战时所产生的感受，表示我们做了不聪明的、伤害到其他人的事，证明我们之前的想法是错的。"

回到技术领域，如果你知道大家正在寻找的问题的根源在于你的程序，那么请你主动站出来，简要地解释一下存在什么问题，出现问题的原因，以及你能想到的解决对策。在专业领域，出现这种讨论的原因，往往是由于主导者在情况变得糟糕时得不到直接的答复。如果能够及时向他们提供信息，说明问题出在哪里，正在采取哪些措施以最大限度地降低问题再度发生的可能性，他们就能对问题的影响做出准确的评估了。

敢于承认错误，并积极地弥补错误所造成的损失，是一种非常可贵的精神，值得我们大力称赞。这让我想起爱迪生的一位学徒，不小心打碎了一天的劳动成果，实验室其他人都在责备他的粗心，只有爱迪生不仅没有打击他反例安慰他，并让他继续担任成果传递的工作，这一经历最终造就了一位科学家。如果哪一个领导因为你主动承认错误，而把责任全部推给你，那你也应该离开他了。

2.2 组建团队

"一流人才招聘一流人才，二流人才招聘三流人才。"

——史蒂夫·乔布斯

管理者首先必须学会管理程序员和软件团队的技巧，换言之，必须学会了解员工，包括如何聘用他们、激励他们，进而领导他们开发并交付杰出的产品。

2.2.1 招聘策略

1. 了解岗位需求

在招聘之前，我们需要明确知道自己需要什么样的程序员，需要分析自己所提供的岗位，即便都是合格的程序员，也需要根据我们的实际需求确定最适合的那一位，也就是说，明确我们当前是需要开发经验，还是需要对技术有钻研热情的人，即你是需要：

- 一个可以领导整个团队开展各种实际工作的程序员？
- 一个可以找出产品里隐蔽难寻的设计缺陷的程序员？
- 一个具备大局意识、能预想到需求可以如何分解为模块和组件的设计师？
- 一个能主动行动、能很好地配合管理层的工程师？

还是需要：

- 在短时间内可以编写数千行代码？
- 能够快速开发出对客户非常重要的原型系统？
- 能快速领会业务流程，围绕根本需求设计产品。

上面提到的这些特质，它们互相并不是排斥的。但前一类程序员可能是经验丰富的老手，后一类更可能是充满热情的新手，关键看你的岗位、行业需要怎么样的人，招到错误的人，不仅是对的失误，也是这个人的失误，他会觉得自己到了错误的岗位上。

2. 编写职位描述

当我们了解清楚自己的实际需求后，我们需要针对职位进行描述。

职位描述一般包含以下三部分内容：

- 基本信息，包括岗位、部门、直接领导、状态、工作地点；
- 职位概述，包括工作职责和预期表现；
- 岗位最低要求。

具体案例如下所示：

职位：服务端开发工程师

部门：C++ 程序设计

直接领导：C++ 团队负责人

状态：全职

工作地点：杭州

职位描述：入门级职位。负责代码开发、转换、验证与维护。负责根据已有的文档和代码质量标准，撰写风格良好的源代码。能较好地融入团队，并遵循团队管理者和资深团队成员的指导。能在团队管理者的指导下开展工作，并就出现的问题主动进行沟通。

职位要求：

- 具有研究生学历，211 大学及以上学校优先。
- 掌握 Windows、Mac 或 Linux 系统中的任一种，掌握两种及以上优先。
- 掌握 C/C++ 及其调试技术。
- 有良好的编码实践，对基本的计算机科学原理有初步的认识。
- 对互联网技术和通信协议及其相关技术有所了解。
- 对数据库技术有所了解。
- 对多线程技术有所了解。
- 能适应团队工作，能根据指导较好地完成工作。

- 能够自我激励，具有较强的学习能力。
- 能够配合团队管理者制订工作计划，并预估工期。
- 能够根据工作的变化做出相应调整。

建议不要只写一份职位描述，而是要多写几组，以体现对不同工作的能力要求。根据过往经验，多写几组职位描述可以更容易让应聘者看到职业发展和晋升机会，不仅有利于 HR 招聘，也能帮助应聘者定位。大家要记住，招聘一位员工，从一开始就已经使用了公司的资源，HR 招聘专员、面试官、会议室、领导、HR 薪酬专员、HR 合同专员等，这一系列的人员、物资都是公司的资源，所以一定要确保招聘策略准确无误，避免出现对的人看到我们错误的岗位描述。

3. STAR 面试法

STAR 是 SITUATION（背景）、TASK（任务）、ACTION（行动）和 RESULT（结果）四个英文单词的首字母组合。

在招聘面试中，仅通过应聘者的简历无法全面了解应聘者的知识、经验、技能、工作风格、性格特点等，而使用 STAR 技巧则可以对应聘者做出全面而客观的评价。

- 背景（SITUATION）：通过不断提出与工作业绩有关的背景问题，可以全面了解该应聘者获得优秀业绩的前提因素，从而获知所取得的业绩有多少是与应聘者个人有直接关联的，有多少是与市场的状况、行业的特点有关的。
- 任务（TASK）：每项任务的具体内容是什么。通过这些可以了解应聘者的工作经历和经验，以确定他所从事的工作与获得的经验是否适合现在的职位。
- 行动（ACTION）：了解他是如何完成工作的，都采取了哪些行动，所采取的行动是如何帮助他完成工作的。通过这些，可以进一步了解他的工作方式、思维方式和行为方式。
- 结果（RESULT）：每项任务在采取了行动之后的结果是什么，是好还是不好，好是因为什么，不好又是因为什么。

4. 招聘文化

好的招聘流程可以促成好的工程团队文化，必须做到快速回应、测试驱动和积极沟通。

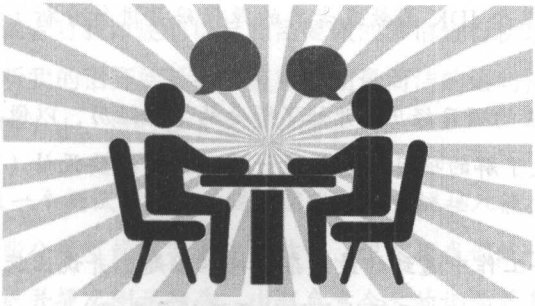
如果你在面试中碰到好的候选人，在面试时你需要回复他们：“你最迟在两天内可以收到我们的通知，如果没有，请直接给我发邮件。”如果候选人在某个领域有很大的影响力，那么就没必要花两个礼拜的时间才把他们招进来。如果动作慢了，你就有可能错过优秀的候选人。

如果你把代码看得比人重要，那么你就无法真正建立起一个技术团队，即使候选人接受了录用通知书（Offer）也无法改变这个事实。

管理和组织流程的最大问题来源于价值的不对称，也就是说一套做一套。重点不在于招到“最聪明最优秀的人才”，你必须明白，你在招聘过程中的表现在一定程度上体现了你的团队价值观，所以请无限高度重视招聘。

记住，招聘是一项长期且繁琐的工作，很多时候没法完全依靠 HR，需要自己花费大量的时间和精力去寻找和沟通。即使如此，仍会有一些岗位长期空缺，找不到合适的人选。招聘一个技术人的时间、金钱、精力，以及培养的成本都是巨大的，如果留不住，就等于前功尽弃。

2.2.2　面试技巧



优秀的主管要有足够的判断力，学会挑选能完成自己目标的优秀人才。

1. 学历

对于中小型公司来说，不要太在意学历，学校经历并不是关键因素（看看史蒂夫·乔布斯和比尔·盖茨）。对于工作很多年的人来说，他的实际工作经历远比大学学历重要得多，因为当一个人离开学校的时间足够长之后，学位就基本没有意义了，经验和能力才是最重要的，学历仅仅是在大学毕业时证明你的学习能力不错，一旦进入公司后就只看你的成绩了。

2. 面试题

我比较喜欢自己出面试题，不喜欢网上千篇一律的题目，当然公司出的笔试题我也会认真批改，并将它作为挑选面试者的依据之一。

我们以 Java 程序员面试题为例：

1）请使用两种设计模式编写代码（考察基础编程能力）。

2）请描述面向对象的三个特性及使用场景（考察面向对象的基础知识）

3）编写数据结构相关的程序，例如“实现一个特殊的栈，在实现栈的基本功能的基础上，再实现返回栈中最小元素的操作”（考察对于数据结构的了解）。

4）编写算法设计题，例如“给定一个 32 位整数 n，可为 0，也可为负，返回该整数二进制表达中 1 的个数”（考察对于复杂算法的了解）。

5）请描述 JVM 堆、栈、方法区的用处和区别（考察对于 JVM 的了解）。

6）举例说明 Java8 与 Java7 的区别，写出对比代码（考察对于新技术的了解）。

7）对您使用的 GC 作原理性描述（考察对于垃圾回收机制的了解）。

8）举例说明某个 JDK 函数或者类的源代码并进行分析（考察对于技术原理的了解）。

9）说说您过去的项目经历（开始了解他的工作经历，以便后续进行提问）。

10）画出您最了解的项目的总体设计图，或者概要设计（了解他的架构设计能力）。

11）说说过去工作中遇到的最复杂的技术难题，并说出您的解决方案（考察他的问题解决能力，也可以了解他所做的项目的复杂度）。

12）工作中您最讨厌领导的哪些行为？过去的经历中有没有发生什么不愉快

的经历（了解他对领导的期望）。

13）为什么离开上一家公司（可以深入聊聊，了解这个人的个性）？

14）反问他有什么问题（别小看这一题，一般来说，如果他有自己的职业规划，他一定会问你很多关于你们公司、岗位的问题，或者如果他对这个岗位很感兴趣，他也会问。反之，如果他想都不想直接说没问题，那我觉得应该多和他聊聊为什么会来面试，仅仅是为了有一份工作吗）。

我从2015年校招开始使用自己的这套面试题（题目内容会变化），效果还是很明显的，一些笔试题做得很好的学生，到了这一环节就开始暴露出问题。我记得在西安的一次面试，一位学生已经拿到了阿里A等实习评价和Offer、华为特等Offer，以及网易和其他一些互联网公司的Offer，所以她过来面试时其实是带着不屑的，加上等的时间比较长，很不开心她进来后，很不情愿地开始了面试问答。我知道并理解她的情绪，没有和她正面冲突（这是个人的职业修养，能够控制住自己的情绪，对于一名技术管理者来说是核心能力）。等技术交流完毕，她了解了我们的技术背景后，我开始和她聊Offer、聊未来、聊技术情怀，最后她鞠躬离开。

面试过程体现了一家公司的技术能力、思维和管理能力，绝不可以轻率应付，你代表着公司，而不仅仅是你个人，如果你不够资格，或者根本不想做好，那请你让开位子，请合格的人来坐。

3. 逻辑思维观察

我一般会让面试者对我口述的场景和需要做的事情进行快速地归纳总结，从他开始阅读或者倾听我的话开始计时，直到他用最简洁的语句描述出事情的背景及我希望他做的事情为止。我们看下面这个例子：

“我们今天想做一个测试，准确地说是想做一个系统接入能力的测试，也可以说是性能测试，有些公司可能也有其他的称呼，但这里我们统一叫性能测试，需要在三周时间内完成并提交测试报告。小赵负责今天的记录并参与测试过程，小张负责做一个计划，或者说方案，就是说清楚我们具体应该按照怎样的思路来完

成测试，以及我们需要输出的结果。请小张记录，我需要拿到系统的接入能力数据，最好是每秒的 TPS，还需要观察一些其他与当时运行状态相关的机器状态信息，比如 CPU、内存消耗、磁盘 I/O、网络 I/O 等，你自己再考虑一下还有什么我没有提到的，具体的测试方案还包括客户端和服务端的测试方案设计，这些我们会后会进行单独讨论，任务需要在本周五完成并提交评审，下周一完成评审。小王负责根据方案出一个用例，你需要等小张的方案评审完毕之后开始做，不过你可以自己先思考起来，具体可以去参考一些外部公司的性能测试用例，我周五来找你到时会结合小张的方案进行具体讨论，这个任务需要在下周四之前完成，下周六完成评审。所有评审完毕后，我们一起商量后续的具体测试分工，测试报告输出时间不变，三周之后，具体分工下周安排。”

我不喜欢拖泥带水、啰嗦的人，喜欢做事干练的人，因此我针对这道题目比较倾向的总结（会议纪要）是：

1）决定进行性能测试，三周时间，需要提交性能测试报告；

2）参与人员：领导、小张、小王、小赵；

3）小张负责测试方案编写，重点是拿到系统接入能力（TPS）数据及观察机器状态信息，本周五提交方案，下周一完成评审。领导会后单独找你讨论；

4）小王负责测试用例编写，请先自查资料、准备内容，周五根据测试方案编写用例，下周四完成编写，下周六完成评审，领导周五会找你单独讨论；

5）剩余一周半时间进行测试并输出报告，具体分工领导下周安排。

4. 心理状态观察

面试时，我们可以通过提问的方式了解面试人为什么离开上一家公司，是不是发生了严重的冲突，或是存在隐忍的矛盾。如果你不善于观察一个人的心理状态，可以请 HR 协助一起面试，HR 一般都受过专业训练，可以在短时间内判断某个人的心理状态和性格，这些都有利于你招聘。

5. 面试官形象

公司派我们出去面试，代表的是公司的形象，这自然不用多说，另外需要注

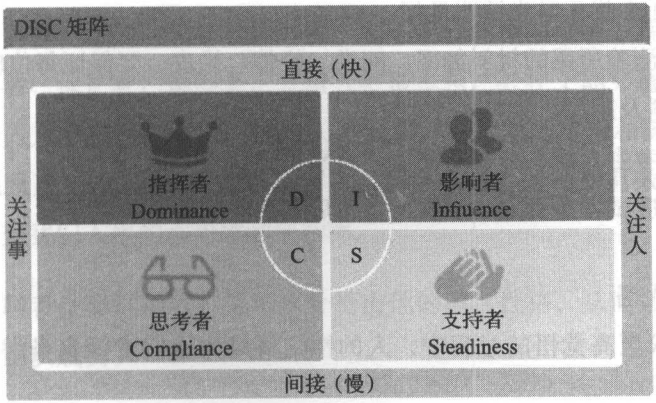
意的是，你面试的人是在整个公司使用的，不是只给自己使用的，所以需要对技术严格把关，加强责任心，只录取技术能力过关的、性格适合做技术工作的人，不要受自身情绪影响，导致录取不合适的人。

2.2.3　性格分析

首先我们应该明白，人的性格分为很多类。内向的人面对问题时喜欢一个人独立思考、整理思路，我就是这一类型的人，我不喜欢一大堆人围着我讲。但是外向的人通常很喜欢跟大家聊天，聊得越多，整理得越清楚，他们喜欢从几个发散的点开始讨论，在你快晕头转向时，他却能画出一张清晰的图。

有的人是判断型，这类人非常喜欢按计划办事，另一种人是认知型，非常喜欢自由，不按规矩办事。工作中，项目管理需要判断型的人，因为他们对时间的把握很精确，总是能够按照时间完成。认知型的人不喜欢规划，十个工作日，前面八个工作日没干活，因为没有感觉，最后两个工作日才开始工作。但是认知型的人也有发挥的时候，在系统出现重大故障时，喜欢计划的人一下子懵了，不知道怎么办，而对于认知型的人来说应对这一类突发事件就是他的强项了。当你对人的性格有所了解之后，你才知道跟不同性格类型的人沟通需要用不同的方式。我们通常喜欢用自己的方式去理解其他人，而真正要用别人的方式来理解他其实非常难。

1. DISC 个性特征理论



1928年，威廉·马其顿博士出版了《正常人的情绪》一书，书中提出了DISC个性特性理论。他认为人类行为是个体自身的反应（主动或被动）以及其对环境的认知（友好与敌对）相互作用的结果，以这两方面为基本轴，可以区分出个体与环境互动的四种典型模式，每一个个体或多或少会呈现4种不同的模式：

- 支配型（Dominance）：在敌对的环境中保持主动的态度和反应；
- 影响力（Influence）：在友好的环境中保持主动的态度和反应；
- 稳定性（Steadiness）：在友好的环境中采取保守的态度和反应；
- 遵从性（Compliance）：在敌对的环境中采取保守的态度和反应；

管理行为作为工作情境下的一种特殊行为，它会受到人格特征的影响。具有不同人格特征的个体在同样的工作情境下会表现出不同的管理行为，往往会在工作中形成自己的管理风格。DISC个性测验就是把个体安排在这样的一种管理情境中，描述个体的优势、在工作中应注意的事项以及一些个体倾向等。例如，如何影响他人、对团队的贡献是什么、什么时候处于应激状态，这些能使个体更加清楚地了解自己的个性特征，企业也可以有针对性地考察应聘的个体是否具有对企业和职位十分关键的人格特征，以此作为筛选人员的标准之一。

DISC个性测验不限定时间，整个测验大约需要十分钟左右。纸笔作答或计算机测试均可，也可以集体测试，具体过程如下：

1）依据预定的参试人数选择适宜的测验地点，布置考场。考场环境应保持安静整洁、无干扰，采光照明良好。

2）准备测验所用的材料如下：问卷、铅笔、橡皮，要保证每位应试者都有完整的测验材料及用品。

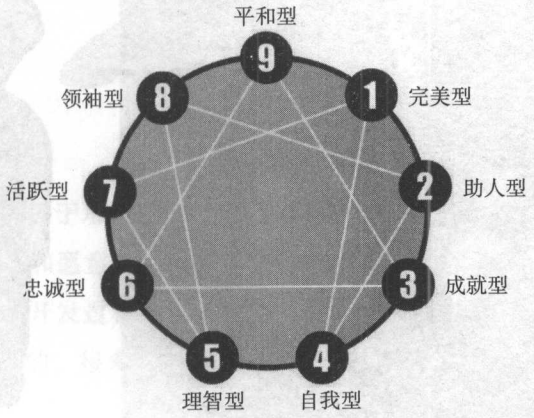
3）安排考生入场，并宣布测验注意事项。

4）检查应试者是否完成了所有题目后，回收问卷，测验结束。

2. 九型人格

人在追求至高觉悟的过程中，人的性格将成为他们发掘自身潜力的引导者。

人性的发展是一个包括了不同阶段的完整体系，从最基本的性格特征，到一些不平常的潜能，比如爱的能力、感受他人的能力和先知先觉的能力，这是一个漫长的演变过程。



口头交流是获取九型人格信息的最好方式。让属于同一种性格类型的人坐在一起，讨论他们的生活，可以帮助他们更好地了解自己。一群外表迥异的人在经过一小时的交谈后，会变得非常相似。旁观者能够从他们的身体姿态、情感表达、面部表情，还有他们散发出的个人气息等细微信息发掘他们之间的相似性。当交谈者的性格特征逐渐显现出来时，观察者会明显感到，每一种性格类型都有一种独特的感觉，一种与众不同的气质。

九型人格是一种深层次了解人的方法和学问，它按照人们的思维、情绪和行为，将人分为九种：完美主义者、给予者、实干者、悲情浪漫者、观察者、怀疑者、享乐主义者、保护者、调停者。九型人格最卓越之处在于能穿透人们表面的喜怒哀乐，进入人心最隐秘之处，发现人最真实、最根本的需求和渴望（这个对于工作来说很重要）。九型人格能够帮助我们洞察人心，从而用有效地方式应对他人，最终提升我们人生的幸福感和成功率。

通过让应聘者做测试题目，你可以分析出他的内在性格，从而帮助你决定是否接纳他成为团队成员。

2.2.4 人员分类

1.“独狼”和“农民”

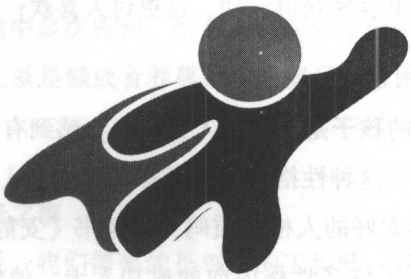
我们可以把程序员分为“独狼”和“农民”这两类，“独狼”更喜欢一个人完成工作，也就是说，当问题出现时，他们的第一反应是独自去解决问题。他们常常跳过规划阶段，最终得到的是一次性解决方案。

从团队角度来看，我们更加希望软件开发像种地一样。农民会有条不紊地先去了解地形、研究土地的化学成分，然后种植、浇水、除草，最终收获粮食。可靠、可扩展、可维护的软件也都是这样有条不紊地开发出来的。

许多“独狼”都是优秀的程序员，但是你需要对他们进行“贴身”管理。他们有着当主角和引起团队内部纠纷的倾向和能力，我们需要做的是尽量多地关注他们的工作，了解他们的具体工作内容，如果出现问题要马上采取措施进行纠正，否则最终局面可能会失控。

只能当“独狼”的程序员不会在任何一家企业待太久。要么是你对他们总是自顾自地向前冲感到厌烦而辞退他们，要么是他们因长期受限制感到厌烦而主动辞职。

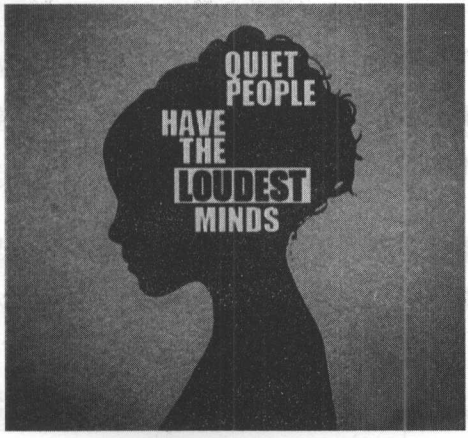
2. “英雄”



这一类人指的是勇于承担需要付出极大努力才能完成的任务，并最终取得成功的人。在承受压力和愿意付出方面，“英雄”和“独狼”有点相似，但“英雄”更能够在团队工作和开发过程获得成功。此外，“英雄”是团队内部培养出来的，不是从外面雇佣过来的，很多时候他们会在企业的成长为超级明星。

管理“英雄”具有一定的挑战。你如果总是希望他们付出超人的努力，时间长了后会发现使用过度，从而发生矛盾。此外，作为技术团队管理者，你也需要有一定的技术功底，能够让“英雄”觉得可以从你这里学到一些知识，这样才能合作愉快。最后，需要对“英雄”的薪资、福利有所倾斜，并且多和他们沟通，让他们自己选择技术方向，为他们指明职业发展方向，并把他们放入关键项目里。

3. 内向的人



外向的人天生善于交际，长袖善舞，也更容易引起别人的关注，在团队中表现更加活跃，所以他们更容易获得成功，也更讨人喜欢；内向的人不容易得到重视、关注甚至快乐。

有些父母发现自己的孩子是内向的人，他们会感到有点遗憾，甚至难过。还有一些会尝试去改变孩子这种性格特点，更有甚者，会认为孩子有心理障碍。内向型人格真的是一种不太好的人格特质吗？畅销书《安静：内向性格的竞争力》的作者苏珊·凯恩列举了许多性格内向的成功人士，他们安静、稳重、深思熟虑，默默地带给这个世界许多改变。如果他们不幸被强制改变成外向的人，这个世界会少许多天才，多出来许多“病人”。

如果一个内向的孩子在成长过程中总是被父母纠正其性格，被同伴嘲笑或者排斥，那么他就会对自己的性格产生自卑情绪，不仅不会去寻找自己这种性格的优势，反而还会痛恨自己为何是这样的人。这种自卑会给他内心造成很多冲突，这些冲突会引起他的心理障碍，比如会引起社交恐惧症、焦虑症等。

内向的人表现为非常沉默、内敛，几乎让人感觉不到他们的存在。他们可以把工作完成得很出色，但是对团队执行力或者在会议上几乎不会有什么贡献。他们在一对一的时候能正常进行交流，一旦退回到人群里就跟消失了一样。

在会议上让他们发言时，当他们分享自己的意见或见解时，都要给予正面的支持，这样可以逐渐帮助他们建立自信，让他们感觉自己对于团队是有贡献的。找机会跟他们交谈，当面认可他们的贡献。与他们的交流要单独进行，通过一些小事情与他们建立特殊的联系，例如分享工作经验、交流教育孩子心得等。总之，要想方设法与他们建立更紧密的联系。

4. 带有明显负能量的人

尽量避免团队里存在充满负能量的人，他们可能会挑拨离间或发泄不满情绪来影响整个开发团队，并且对组织造成严重破坏。如果没有他们，有些负面情绪可能永远不会出现。比如“团队管理者根本不关心大家”，带有负能量的人会进

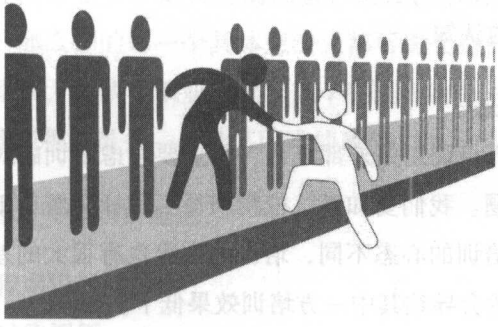
一步夸大事实，把一些细枝末节的事情说成是管理层故意针对程序员的行为。

任正非在内部讲话中多次提到，所有人都应乐于到一线工作，不要总是给自己找很多借口，实际上就是懒或者养尊处优习惯了，有困难就解决，发牢骚、推脱，并无任何意义。

5. 离奇之人与离奇之事

对于技术团队来说，我们需要的是刻苦的工程师，需要的是能够解决问题的工程师，而不是搞办公室政治的高手，或者腹黑的工程师。不管是一心想当“老大”，不择手段压制同事的人，还是对同事很粗鲁的人，或是到处借钱的人，早点让他们走，无论他们的水平有多高，都不要犹豫，也不要期望自己能够改变他们什么。让这些人尽快远离团队，这样整个团队都会轻松很多。

2.2.5 新员工入职



做好新员工入职工作非常重要。这是一个向新员工表达你对他们非常看重，且你的团队管理与运行状况也十分良好的机会。

在大多数公司里，团队管理者一般都要负责为新员工寻找工作定位。我们需要考虑团队现有的人员配置，找到与新员工个性相近的老员工，另外还要考虑到新员工的工作职责，以及团队领导、技术领导和架构师的座位。

我的习惯做法是带着新员工逐一认识团队成员，自己带着他去食堂吃午饭，

并指定新员工的导师（技术上的），然后和他聊聊最近一个月准备给他安排的工作等，等他和团队成员熟悉后，我会找机会组织全组人一起吃顿饭，一起欢迎新员工。

新员工很容易流失，大家要引起注意。新员工流失的原因大致有三类：

（1）没有对新员工进行明确区分

校招新员工和社招新员工完全不同：校招新员工就像是一张白纸，太多无经验、能力不足，而社招新员工则正好相反，因此在进行培训时要将两者区别对待。实际上，从更专业的角度讲，需要按照工作年限、岗位级别等区分社招员工的培训内容和方式，对不同的人做正确的培训，会对后续的工作帮助更大。

（2）忽略新员工直属主管的重要作用

企业招募新员工都希望其能为企业创造价值并长期为企业服务，但人才成长是一个长期培训的过程，任何企业都无法通过新员工培训就帮助其快速成长，还要依靠新员工主管后期对新员工进行更多的培训才能达成，只是很多企业的主管在这方面没有明确的认知。

（3）培训方式缺失

无论是选择内部培训还是外部培训，重点要考虑培训的对象和背景，这正是很多企业忽略的问题。我们要知道，校招新员工和社招新员工由于其背景和经历的不同，他们面对培训的心态不同，培训的内容也有很大的差异，如果用相同的方式进行培训，必然会导致其中一方培训效果低下。

2.3 管理团队

大多数公司内部都是垂直组织结构，所以管理团队、管理自己的团队领导岗位，大致上可以分为向下管理、向上管理、对外管理、自我管理这四个方面，管理的最终目标是：“不要让你的下属陷入困境，不要让你的同事陷入困境，尤其是在任何情况下，都不要让你的上级陷入困境”。

如果你想要建造一艘大船，不要立马号召大家开始收集木材，也不要立马分配任务和工作，而是应该先教会他们去憧憬广阔无垠的大海。

——Antoine de Saint-Exupéry[⊖]

2.3.1 向下管理

向下管理包含的内容很多，下面我们分别讲解。

1. 技术尊重

要了解程序员，首先需要深入理解他们使用的工具、流程，以及程序设计的艺术。你理解得越深入，在和他们进行技术对话时，参与能力就越强，越容易获得他们的尊重。微软的一位程序架构师这样评价比尔·盖茨："盖茨最喜欢和他的程序员一起将程序分析到比特、字节层面。在技术战斗中他可以非常轻易地守住自己的阵地，他之所以可以获得程序员的尊重，因为他可以轻易地战胜他们。"

成功地管理程序员最重要、最关键的因素，是在技术层面得到他们的尊重。如果没有技术尊重，那么你的每一个具体想法，都可能会遇到主动或者被动的阻碍。正是由于这个原因，那些在职业生涯的某个时期没有做过程序员的团队管理者，会觉得有效地管理程序员是一件极其困难的事情。

要得到技术尊重，关键因素包括：

- 理解计算机程序设计的艺术
- 拥有良好的过往履历
- 做出过技术贡献
- 追逐技术潮流
- 成为一个技术或者职业组织的活跃成员

⊖ 即圣埃克苏佩里（1900-1944），法国作家。他是法国最早的一代飞行员之一。1940 年流亡美国，1943 年参加盟军在北非的抗战，1944 年他在执行第八次飞行侦察任务时失踪。其作品主要描述飞行员生活，代表作有小说《夜航》，散文集《人类的大地》、《空军飞行员》，童话《小王子》等。

❑ 展现出强大的个人价值

这些要素解释了为什么从公司外部招聘过来的技术团队管理者很难真正落地（短时间内开展有效的管理工作）。你所选择的这位空降管理者，你需要充分考虑他是否有良好、可以被证明的履历，这样才能让他获得团队的尊重。所以说，一般情况下技术团队是不会空降高管的。

2. 强化现有的团队

我们假设现有团队全部由普通程序员（能够完成交代的工作，但是没有主动创造能力）组成。我们需要做的是招聘 1 ～ 2 位杰出的程序员，这一步要耐心，明确候选人是否真的是杰出程序员，因为只有招聘到正确的人才能让工作高效，如果招到的员工很差，那么你就没有时间去处理其他的工作了（总是会有各种问题困扰着你）。

系统程序员 / 架构师往往会在团队里显得格格不入，这是因为他们很多都是“独狼”，他们可能脾气很差，也可能在技术上具有个人主义。和这类人不同，杰出的程序员能够以一种优雅、简洁、易懂的设计来架构大型的复杂系统，这些优秀的系统往往能让所有其他程序员的工作都更加轻松。因此，单个人就能带来巨大的杠杆效应。需要让系统程序员 / 架构师参与到开发工作中，不要让他们只设计、不编码，应该把他们引导成为杰出程序员。

3. 团队组成

杰出的程序员需要一群称职的程序员来配合，依赖他们完成日常的开发工作，实现设计好的系统和产品。和橄榄球类似，一个杰出的橄榄球队中必须要有那些负责阻拦和抢断的队员，而一个杰出的开发团队则主要由称职的程序员组成。这让我想起了电影《冲锋陷阵》[⊖]的最后一幕，四分卫拿着球向对方阵地冲去，周围一群队友阻拦对手球员，能力弱的以自己的身躯直接和对方一对一硬拼，能力强的干掉一个又一个对手，直到四分卫冲过对方底线。这部电影我看了

⊖ 又叫《光辉岁月》，丹泽尔·华盛顿、威尔·帕顿等主演。

很多遍，也是我带领团队的精神指导，每次看到这一幕，我都会非常激动。

4. 薪酬组成

每个人都是独特的，薪酬是由下面这个复杂的公式决定的：

薪酬 =i* 工资 +j* 股票期权 +k* 福利 +l* 与谁共事 +m* 工作地点 +n* 做了什么 +。。。。

每个变量前面都有一个与其关联的系数，系数的值会影响公式的结果。每个人都应该考虑对自己来说哪些因素会比较重要、有多重要，然后在 0 到 1 之间为系数取值。

5. 进度管理

我有一块小白板，我把它放在自己的面前。每天早上我都要写上今天需要参加的会议、自己要做的事情，此外，每天上午半天时间我会和每一个项目（产品开发、预研、调研等）的团队成员过一遍当前进展。大家坐下来，好好谈谈已经实现的设计或代码，对疑惑、问题进行讨论。因为这种方式可以确保自己不仅仅依赖于状态报告、项目时间表，这种方式也可以让你能够接触到说真话的员工，他们会告诉你哪些地方做得不够好，并且会主动请求团队管理者帮助，而不需要团队管理者来催促他们。最高效的团队管理者往往都是坦率的，也往往对下属有足够的时间，能让员工找到他们并说出自己的想法，他们会认真倾听。

从长远看，一些不紧急但是很重要的事情，如果迫于压力被放下，到后面往往需要花费好几倍的代价才能弥补回来，技术管理者需要时刻保持警惕，慎重地做出决策，做正确的事情，要能够为公司的长远利益负责。

6. 效率管理

影响组织效率的关键因素是人，如果每一个人效率提高 10%，组织的效率就会提升很多倍。团队的效率并不是简单的加法关系，而是乘法关系。所以，如果每个人的效率都能够有所提升，对结果的影响是很大的。

回到技术领域，有一种说法，高效的软件工程师，产出是普通的工程师的 10 倍。苹果的设计团队，只用了 20 多个人就支撑起苹果几乎所有产品的设计工作，甚至苹果新落成的环形办公大楼也是他们利用业余时间完成的，其效率令人咋舌。

如果组织中每个人都照章办事，不乱插队，整个组织就是高效的。如果每件事情都有唯一的责任主体，出了问题能够快速找到问题根源，组织就会更加高效。如果流程中每个人都对流程的优化贡献出自己的力量，则组织就会更加灵活，更加高效。

那么如何提高效率？

（1）提高效率，最重要的是先确保做正确的事，再把正确的事做对。因此，先明确目标和结果，以及结果对最终客户的价值大小，这是提升效率最重要的事。在这个基础上，我们会发现很多事情根本就不应该做，而不做就是最大的效率提升。减少一件事情，或者在事情中减少一个步骤，就是最大的效率提升。

（2）所谓效率其实就是性能。性能往往受限于系统的资源瓶颈。我们应该看清楚整个系统中的资源瓶颈是什么，是机器厂房、人力、资金预算，还是周转率，围绕它来组建流程，提升效率。

（3）高效与工具化、自动化程度强相关，因为自动化具有规范、一致、高速、闭环反馈等特性，而且不会因为疲劳而下降。因此高效率的组织，一定会大力发展这些能力。

（4）高效的组织，每个人都会主动优化流程，并且持续改进。

7. 引导工作

团队管理者的一项重要工作是引导事情走向正确的方向，并确保团队成员之间以及与其他团队之间有正确的沟通方式。需要注意的是，引导的最终目的是为了让事情完成，而不是把关注点放在如何完成上。对于一个团队管理者来说，要想最大化地利用自己的时间和技能，就要引导程序员自己做出正确的决定，而不应该自己就把决定做了。这样做可以帮助下属员工培养技术、积累经验、建立自信，还能获得具体执行决定的员工的认同。

如果你发现自己经常需要讨论非常具体的命令以及如何执行，说明你没能很好地利用你的管理技能，或者没能赋予下属员工足够的权利。作为团队管理者，你必须指出大方向，然后做好充分的检查，以确保员工做出正确的决定并实现。应及早检查员工做出的重要决定，否则当你想中途接手并做出改正时，员工可能已经做了大量无用的工作。这也是为什么我在“进度管理”一栏中强调自己每天进度跟踪的重要性。

记住，优秀的主管要有足够的自制力，不在下属工作时瞎指挥。

8. 保护成员

做过项目的团队管理者一般都有这样的经历，团队成员正在专心处理现场问题，然后莫名其妙被人投诉，投诉可能来自市场部门，也可能来自技术支持部门，或者研发部门。也容易出现团队成员每天被大量无用会议烦扰，不去的话就要被投诉，这类情况在大公司司空见惯。

我们要学会保护团队成员，让他们免受组织中的各种问题、争议和“机会”的干扰。在一些稍大的公司内部，官僚主义会通过各种文书工作来忽略或者缓冲每天的各种请求和问题。在小一些的公司里，面对的挑战是各种销售驱动的机会、客户驱动的争议问题，以及管理驱动的想法，你作为团队领导者，可能是他们最后或者唯一的防线。

很多团队管理者的大部分工作内容是处理这些问题，如果让你的下属去处理这些问题，他们最终可能会被淹没在这些繁杂事情的洪流之中，进而极大地降低他们的程序设计效率。通过保护你的员工，可以让他们避免把宝贵的时间浪费在临时事务上，而且也能让他们工作得更愉快，因为某些问题可能会演变成完整的流言，让人担心项目变更，担心收购、重组或裁员，这些流言会大大降低士气。

此外，你应该保护你的员工免受开发之外的同事或者部门的攻击。这些攻击或者抨击往往没有充足的信息或事实根据，有些抨击是出于好心的，有些则是恶意的，甚至可能包括个人攻击。我的建议是对这种情况保持警惕，处理之前先充

分了解情况，如果确实存在无根据的情况，事发当时或事后私下沟通，告诉那个进行抨击的人，指出他的行为是不恰当的，让他知道事情的严重性，而不是一味指责自己的员工做得不够好。

有一种情况我想特别说明：公司内部非重要部门组织的会议，尽量不要放在周五的晚上和休息日进行。利用这种休息时间，看起来很有效率，其实是在过度使用研发资源，过度消费员工对公司的满意度。周五晚上和休息日可以打扰研发人员的事情是：

（1）现场问题，必须立即处理（不处理会损害公司未来利益）；

（2）重要客户提出需求，要求立即做出回复（不处理会损害公司当前利益）；

（3）特别重大的突发事件（对你和公司都很重要）。

9. 评估和改进绩效

“信任但要核实”，这是里根总统经常引用的一句谚语，也是列宁的口头语。

团队管理者最重要的职责之一是评估员工的工作绩效，并持续改进他们的绩效。每日反馈、季度 / 年度绩效审查，以及每月或者每季度目标等都是很好的巩固，可以帮助你完成绩效的评估和改进工作。

比较直接的方式是为每一个人设定工作目标并规定完成的时间，接着定期审查这些目标的进度和实现方式。

10. 任务责任制

每项任务都必须有且仅有一位负责人，如果有两个负责人，那就没有人负责了。开发经理的职责是确保为每项任务指定负责人，而不是亲自去完成每一项任务（开发经理可以指定其中的某一项工作由自己直接负责）。应当明确每项任务，确保为每项任务指定一个负责人，推进任务。还要定期检查以下三个问题：“是否清楚整体的目标？是否清楚你的任务对实现整体目标有怎样的贡献？对于你所负责的部分，有哪些东西妨碍你达成目标？”我自己的做法是，只有担任责任人的员工，才能在考核中得到良好或优秀的评价（前提是把事情做好，做不好就要

承担责任），没有担任责任人的员工，最多只能给予合格评价。

我们实行了任务责任制，这种长期责任，不是我们的管理保守了，而是在内、外合规的条件下，鼓励在集体主义下的个人更好地发挥。我们呼唤英雄，也要宽容英雄的一些过错。英雄要更加自律，因为天降大任于斯人也。

11. 裁员

表现差的人给团队拖后腿的方式很多。他们占用了预算的一部分，但是却无法交付有效的成果。其他人如果看到他们差劲的表现，也会失去动力或者失去对你的尊重。表现差的人可能会影响项目的进度，从而对项目中的每个人都产生不利影响。因此无论如何，这种情况必须尽快解决。

通常，终止合同是一个对双方都不错的选择。表现很差的员工都知道自己很差，他们每天上班和睡觉都背负着这个沉重的负担。对大多数人来说，负担会沉重到难以忍受。所以，当你终止合同时，员工通常会感到一种放下包袱的轻松。很多人会选择离开，也有一些人会选择尝试改进，如果选择后一条路，你一定要定期与这个员工会面，审查绩效情况并讨论结果。

12. 主动沟通

承担并顺利地完成与员工的主动沟通，这是你的职责，千万不要轻视沟通能力，这一能力对你的晋升起决定性作用。

两个人只有面对面坐下，看着对方交谈，沟通才不会有障碍。如果你有团队成员在外地或者国外，作为团队主管，你应该主动去拜访他们，并且坚持在那里待一段时间，第一次拜访越早越好。

团队越来越大时，信息之间的传递会变得越来越困难，技术管理者要想搞清楚项目执行的进度、团队成员存在哪些诉求和不满，想要打造一个有战斗力的团队，沟通无疑是最重要的。

我们可以通过配置 HRBP 的方式来加强和团队的沟通，也可以通过要求团队

Leader 每两周组织一次与小组成员的一对一沟通，并在每个月底的时候召开一次全员例会，内容可以是欢迎新入职的同学、总结本月重要项目的进展以及存在的问题、介绍下个月要做的重点工作等，让大家对工作有一个整体的认识，并清楚知道自己在全局中所发挥的作用。

13. 被动沟通

如果一名团队成员在你不太方便的时间来找你聊天，一定要先把手上的工作放下，专心跟他交流，他可能想鼓起勇气告诉你一件大事。聊天的内容可能很简单，比如缺少完成任务所需要的相关资料或者技能，也可能很重大，比如即将离婚、家人病重或者其他对他个人有重大打击的事情，而这些事件对你的工作时间表都有重大影响。如果你的团队成员知道自己受到了最高优先级待遇，那么在事情将变得很糟糕的时候，他们来找你交流的可能性就更大了。

14. 文化冲突

美国有这样一句谚语“吱吱响的轮子先上油。”这句话通俗点讲就是：“有问题或者困难要让别人知道，才会帮助你，会哭的孩子有奶吃。”日本有句类似的谚语：“突出来的钉子先挨敲。”这句话通俗点讲就是：“遇到问题就立即喊出来的人，要被教育了。”这两句话反映了美国人和日本人对于困难的理解差异，所以你要让两个不同文化的民族一起共事，很有大困难。

15. 负面效应

有时候会遇到这样一些问题雇员，他们能写出高质量的代码，但设计方案却很糟糕，或者他们的行为会对整个团队的效率起到反作用。

遇到这种情况，刚开始时可以安排适合他们的工作，如果一个项目不那么重要，可以安排问题雇员承担他能力较弱的工作，问题凸显后你就可以找他进行一对一沟通了，通过事实来指出他的问题，努力让他提高。

16. 会议记录

记住，没有记录的会议，相当于没有开过。每一个重要会议，都需要有完整

的会议记录，包括参加人员、讨论议题（逐一写下来）、讨论过程大体描述、每一个议题的最终结论（包含流程图）、遗留问题、下一次会议时间及议题等。

17. 人才评测机制

正如没有完全相同的两片树叶一样，世界上也没有心理面貌是完全相同的两个人。人与人之间的心理差异主要表现在智力、个性和行为等方面。比如，有的人思维敏捷，有的人想象力丰富，有的人脾气暴躁，有的人性格温和，有的人做事认真，有的人行事草率，如此等等。正因为人与人之间存在差异性，每个人都是一个独立的个体，人才评测才变得很重要。可以说，人与人的差异性是人才测评存在的前提条件。

独特性不是在个体身上偶然表现出来的暂时特点，而是稳定的个人特点。一个人在出生后，经过长期的社会生活，逐步形成对待生活的态度和个人的行为风格，这种特点一旦形成，就不容易改变。比如说，一个性格外向的人，不仅在工作单位爱与人打交道，在社交场合也会是一个活跃分子，不仅今年这样，明年也会这样。正因为个人特点具有相对性、稳定性，才使人才测评变得很有可能。如果个人特点没有这种稳定性，人才测评就没有意义了。

企业太多由许多不同层次的、不同部门的岗位所组成。由于每一个岗位的工作性质、工作内容、技术难度和责任都不相同，所以对任职者的素质要求也不相同。因此，人与岗位的匹配问题就成为现代人事管理的重大研究主题。想要做到人岗匹配，首先需要对人和岗位有客观的认识和评价。为了了解和评价人，就产生了心理测试、面试、评价中心等人才评价手段；为了了解岗位，就有了工作分析、工作描述等岗位分析和评价技术。因此，认识是可以被测评的。现代人才测评技术是基于通过观察人的少数代表性行为，对贯穿在人的全部行为活动中的心理特征和能力水平作出推论和量化分析的一种科学手段。

结构化面试就是首先根据对职位的分析，确定面试的测评要素，在每一个测评的维度上预先编制好面试题目并制定相应的评分标准，面试过程遵照一种客观的评价程序，对应试人员的表现进行数量化的分析。

情境化测试包括多种形式，主要有文件筐或公文处理测验、无领导小组讨论、角色扮演、根据所给的材料撰写报告、演讲辩论、案例分析等。

以某通信公司为例。该公司员工入职时确定员工的岗位级别，该级别和薪水等待遇关联，每一个级别的工资上下限相差 5000 左右。级别从 1 ～ 3 级开始，本科、硕士应届生默认为 1 ～ 3 级。技术级别从初级（8 级）开始，该级别和薪水的关系为弱关联，级别升迁不一定代表加薪水。新人入职后需要参加考试，对应类别的 1 ～ 3 级考试。考试通过后，上传述职材料，组织部门内的技术专家评审小组进行评审，然后对该人做出整体评价。1 ～ 3 级不限制每个技术级别的人数，达到要求即可升级。4 级及以上人员，需要上传述职材料，根据申请评定级别，组织上级部门内的技术专家成立评审小组，对上传材料进行评审，并安排答辩。

每个岗位级别对技术级别有要求，即技术级别是岗位级别的必要条件。部门内部各级资源主管评定员工是否可以升级岗位级别。岗位级别在部门内是有固定比率的，所以存在一个高技术级别的员工无法担任高级别岗位的可能性，但考虑到人员是流动的，这个现象不太会持续很久。员工在当前项目组表现特别出色，给予直接升级，每个部门有小比例名额，以提高员工在项目中的积极性。可能存在 1 个部门的 14 级工资超过另一个部门 15 级工资的情况，公司每隔 2 ～ 3 年会进行工资调整，避免这种情况长期存在。

18. 下放权力

抓大放小，注重结果。

程序员在工作中经常犯的错误是只见树木，不见森林。举个例子，程序员在得到用户需求后立即开始编程，解决用户的实际困难，但由于事先没有进行很好的策划与沟通，客户的需求总是在不断调整，程序员也总是在修改程序以满足客户的需求变化。这样一来，该项目的周期自然就一拖再拖，似乎没有终点。

管理者需要为成果而工作，以结果为导向。一位卓越的管理者不应该在工作一开始就身先士卒地从事具体的工作，更不应该把精力放在研究技术实现的细节

上，而是首先问问自己：“客户期望我做出什么样的成果？”然后再对整个项目进行规划。

19. 激励员工

亚伯拉罕·马斯洛在20世纪中叶提出了需求层次模型。在1954年首次出版的《动机与人格》（Motivation and Personality）一书中，他将该理论引入了商界。马斯洛认为，人的需求可以按层次划分，从最基本的对食物和居所的需求，到最高的追求自我完善，在低层次需求尚未完全满足之前，更高层次的激励并没有多少用武之地。

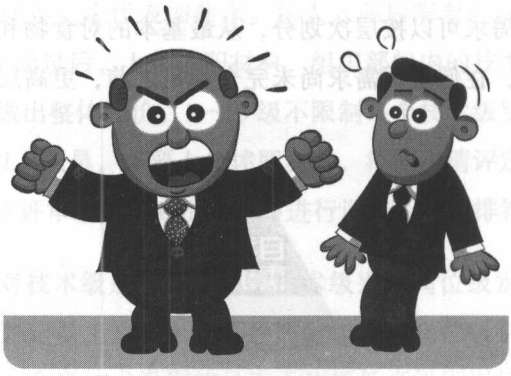
直到今天，需求层次理论依然令人信服，在理解人的动机和个人发展的管理培训中经常会引用它。事实上，马斯洛围绕需求层次理论所提出的观点，今天依然在促使我们不断地改善工作环境，鼓励员工充分发展，自我实现。

我收到过一位网友的提问：“在没有奖金、没有加薪区别的情况下如何激励员工？”针对这个问题，想问他们家老板，连基本的激励都不存在，还怎么让员工做出成绩。

一旦生理需求满足后，人们就会寻求更高层次的满足。佛雷德里克·赫茨伯格在1959年出版的《The Motivation to Work》一书中描述了他认为的激励因素

包括成就、认可、工作本身、责任、发展，只有当这些都得到实现和满足时，人们才能得到真正的激励，这样来看，这些因素意义重大，它代表着更深层次的成就。所以，我们只有更好地了解员工的工作动机，才能更好地激励他们。

2.3.2 向上管理



向上管理其实是四个管理方向里最难的一点。什么是向上管理呢？向上管理指的是如何有效管理你的老板以及你要汇报的那些人。另外，你还需要弄清楚如何汇报、如何沟通，以及要采取什么样的行动，才能让你的老板认为你是一个高效而成功的员工。

管理你的老板看起来似乎是一件比较奇怪的事情，但实际上成功地管理好你的老板可能比管理好你的团队还重要，至少对你个人而言是这样的。这背后的原因在于，成功并不只在于你做了什么，更需考虑别人如何看待你所做的成果。现实中，外在认知往往比实际行动更重要。

1. 了解老板

如何向上管理取决于你是否能够真正理解你的老板，以及其他你可能间接或隐式汇报的人。如果你是向工程副总裁汇报，那么沟通和汇报可能是技术性的，如果你是向首席执行官或者产品副总裁汇报，那么你的汇报可能就要减少技术性，增加信息，也许还要多加一些与产品相关的内容。除此之外，你老板的级别

越高，你的报告就越要精简，越要注重大局，细节更少、局面更大、文字更少、项目符号更多。

关键的是，你需要清醒地评估如何高效地与老板进行沟通。即使是你问他们想要什么，也可能会因为沟通不当，传达给他们错误的信息，或者详细程度不恰当，而他们的自我评估可能也会弄错他们真正想要的东西。所以双方应当一致迭代优化沟通，调整需要沟通的度量。有的事情需要更多的信息沟通，而有的则需要更少。

2. 准备讨论内容

不要把每一个问题都带到你老板那里寻求解决方案。仔细地挑选你的问题，只拿那些真正重要的问题去寻求你老板的帮助。当然，如果你能独立解决问题，而不需要他的帮助就更好了。另外，你还要避免只把问题带给领导的情况，最好是拿着几套潜在的解决方案和问题一起交给他。即使你没有特别好的解决方案，或者没有找到最好的解决方案，也会让领导觉得你已经做好了自己的功课，过来找他是为了寻求他的建议和忠告，而不是直接把问题抛给他。

3. 主动承担

当问题出现时，你也可以尝试主动自荐，帮助他解决问题。

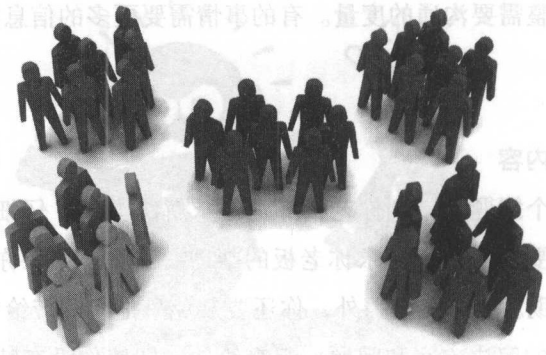
学会给人惊喜与欢乐。了解哪些事情会让你老板时刻关注，并找到办法超水平完成任务，最好是超过他的预期。有时候其实并不是做得越多越好，而是处理较多复杂问题。

你可以非常好地管理团队并完成项目，但如果没有你老板为你护航，你的职业前途必然坎坷。也许看起来不那么明显，但你的薪水、奖金、期权、津贴和机会的多寡，都是由一系列的管理层闭门会议或者你老板自己决定的。

所以一定要主动，做好向上管理，可能会得到更多的职业回报。你损失的只是时间和精力，但能得到的却太多了。

最后，向上管理本身也有直接的回报：当你为更多的交流做出努力时，会潜在加固和上层领导的关系。可能其中一位领导，会成为你的好友，并在你今后的职业生涯中一直支持你。

2.3.3 对外管理



1. 与部门内的人合作

部门内合作的直接效果是，能够让你老板的工作更容易，因为你能解决很多部门内部的事务，不再需要他参与解决。内部合作还能加速项目的完成，因为很多问题和争论，在不被过度关注的情况下，能够更加迅速地得以解决。部门内部合作还能将你和你的同级同事紧密地联系在一起，因为这样你们可以直接共享问题和解决方案，相互鼓励。

这一种合作还可以表现为互相帮助、对遇到的问题互相提供超常规的支持、共享资源和培训，并相互守望。当人们把与自己最亲密的同级同事视为首要援手时，他们更愿意互相协助并帮助对方获得成功。如果你待在一个组织中的时间足够长，那么你的一些同级同事很可能会成为你最亲密的朋友。这种关系应当珍重和培养。

2. 了解其他部门

你要一直保持欣赏其他部门的贡献，因为他们的贡献不管怎么看都和你的部

门一样重要。

当你被聘请或提拔为程序设计经理时，你需要仔细研究组织架构图，找到各个职能部门的主管，想办法让自己逐渐了解他们，或者是各部门中与你同级的经理。请他们吃午饭，或者偶尔停下来与他们聊聊天，提前建立起彼此之间的跨部门纽带关系是很有必要的，以后你真有需要向他们发出请求或寻求帮助的时候会更加容易。

跨部门的纽带关系不仅能帮助你自己，也是促进不同部门团队之间双向协作的重要途径。在跨部门活动中，尽可能成为一个领导者，而不是追随者。你的主动参与将提高你在整个组织中的形象，帮助你在很多看不见的方面取得成功。你在这些活动中花的时间，将会获得大量回报，因为它们可以提高你的工作执行能力。

3. 平级的人相处较难，怎么办

这是网友的一个问题。当你遇到你觉得很难相处的人时，你需要告诉自己，保持自己的职业素养，不要轻易被别人激怒，时刻保持一颗平常心，努力把自己的工作做好。

对于平级的人，如果你确实拿他没办法，我的建议是找你们共同的领导反映你的困惑，或是通过你的领导找到他们的领导，由领导层沟通。对于所有的管理类问题，我觉得首先要自我检查，确保不是自身的问题，然后是沟通，保持高效、简单的沟通，这样可以帮助你至少说出自己的困惑，最后是放松自我，不要被不值得的事情或人所烦恼，过好自己的每一天，全力做好自己的工作，不断提升自我价值，抽出时间陪伴家人，这才是你应该做的。

总结

如果对方是产品经理，尝试以产品的语言去解释。如果对方是计算机科班出身，那么简洁地使用计算机术语，直接说理论的名字或者算法的名字就行（如二次握手、NP 等）。如果对方是其他专业转行的程序员，不妨试试直接拿着代码来讲。如果对方完全不是一个行业的，就尝试针对他了解的一些问题和知识来举例解释。

2.3.4 自我管理

最难管理的人总是自己。我们每个人都善于否认，我们会忽视坏习惯、差劲的执行力，以及对其他人无法容忍的问题行为。要有效地管理自己，有效地管理自己的时间，毕竟每个人一天只有 24 小时，真正的工作时间只有 12 小时左右，首先需要实事求是地评估自己的习惯、实践和行为，然后找出你想要改变的方面，最后实施你的改善计划。

简单来说，个人管理系统就是一个由世界观和方法论组成的整体，构建这个整体的目的是为了实现自我提升，这个整体里包含个人价值观、个人思考模式、做事的流程体系、做判断和决策的方法和一系列的健康生活习惯等。作为一名软件团队管理者，我们需要避免过度管理，做好沟通管理、面对面管理和形象管理。

1. 过度管理

切记，造成延迟和混乱的最主要的原因之一就是允许一个上级直接管理过多的下属。我们自己也要注意这一点，不要过度深入管理每一件事情，或者直接管理每一个团队成员。一般来说，你能够覆盖完整的团队成员，最好不要超过 10 人，多于 10 人以上，可以采用分小组形式间接管理。

随着控制范围的扩大，管理者与下级的交互次数（以及由此产生的指导实践）

呈几何级别增长。这个说法需要考虑到管理者与下级、下级与下级以及管理者与所有下级组合之间的交互，并且假定花在指导上面的时间与交互的次数成正比。例如，你有4个直接下属，在增加第5个直接下属时，完成更多工作的可能性会上升20%，但是交互的次数则可能会从44增加到100，增加了127%。直接管理8个下属可能需要1080次交互，直接管理12个下属可能需要24564次跟进管理。

2. 沟通管理

技术的进步增加了很多沟通途径，且已经远远超过了我们的处理能力。如今，电子邮件、短信、博客、微博、微信、社交网络、Skype、RSS订阅以及其他技术，已经渗入我们的生活当中，而这在几年前是不可想象的。我们已经成为泛滥沟通的奴隶，而这些沟通方式似乎降低了我们整体的生活质量。

要想成为一个有效的经理，你必须建立有效机制来控制和管理洪水般的信息。不能让它控制你。

- ❑ 制定一个切实可行的做法和流程来管理你的工作电子邮件，不能让接二连三的电子邮件分散你的注意力。
- ❑ 别人打电话或发短信给你时，你不必一定答复或回应。
- ❑ 在会议上或在其他需要的时刻，不要让电话或短信转移你的注意力。
- ❑ 委托管理会议议程。
- ❑ 仔细规划你的时间，以减少需求和信息的洪流。
- ❑ 建立或协助建立适当的电子邮件礼仪。

3. 面对面交流注意事项

智慧是您一生从聆听而非说话中所得的奖赏。

——亚里士多德

关注对方，放下你的手机，停止处理电子邮件或写代码，坐下来，看着说话的人。用眼神交流，仔细聆听对方在说什么（并思考他们呈现的信息）。注意，信息并不局限于他们的话语，还有他们的姿势、身体语言、热情或专注的程度。留

意一切信息，通常语言之外的线索反而是最能说明问题的。

谈话时隔着办公桌，或者隔着任何其他东西，都会让人不舒服。

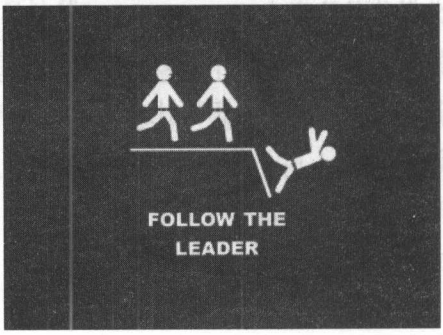
许多人认为管理是自上而下的，具有权威性的，也就是说，管理者处在金字塔的顶端。恰恰相反，建议你把自己放在金字塔的底部，而你的员工在顶端。他们和他们做的工作才是真正重要的。

4. 形象管理

“人靠衣装”其实需要好好思考，如果你看起来邋里邋遢，那么你需要采取行动了，要去克服你的老板和其他高级管理层与你交流时因着装而产生的负面看法。

2.4 影响团队因素

2.4.1 不懂技术的技术领导



不懂技术的人如果做了研发团队的领导，很容易出现严重的问题。例如，技术会议他到底需不需要参加。如果领导是一位技术专家，毫无疑问他需要参加，如果不是，他参加或者不参加，都会引起麻烦，所以尽量避免这样的人出任研发团队领导。另外，对于整个研发过程的管理，不懂技术的人很容易完全从产品角度考虑，忽略研发团队面临的困难和风险，忽略技术人员对于技术的憧憬，造成团队超负荷工作、技术团队缺少技术愿景等情况发生。

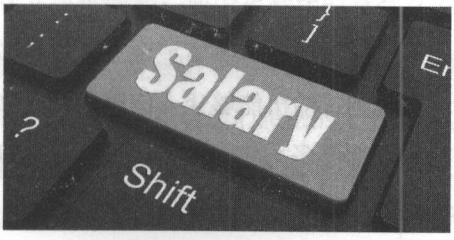
举个例子，遇到业务方提出的需求完成时间点过于苛刻的情况（其实这是一个压力传导问题，业务方收到了客户的压力，本来可以通过向客户解释来减少研发的压力和风险，但是选择直接施压研发）。这时候，你的这位不懂技术的团队老大可能会说，没关系，我们一开始并不需要一个完美的系统，你先上了再说，我们后面有时间再重构和完善（当然有的技术人员也会用“架构和设计是逐步演化出来的”这句话来证明“故障驱动”开发是值得的），这样的想法本质上是错误的。

一些人喜欢将缺少需求分析、技术设计环节解释为“这是敏捷开发，和你们的瀑布式不一样”，这是对敏捷开发的误解，敏捷开发是很好的一套开发流程。敏捷开发的实质是为了解决需求快速变化的情况，需要快速响应需求提出方，快速搭建产品原型用于验证实际效果，而不是说有了敏捷就可以忘记软件工程理论，不管三七二十一，先随便写一堆代码再说，这是不合理的。任何软件工程模型，都不会允许在需求完全不明确、描述不清楚的情况下，开始进行技术方案设计，也不会鼓励在方案设计缺失的情况下开始编码，因为这个时候没有人知道究竟如何编码。

团队领导可以不是对口的专业出身，但是他必须对技术有热情，必须有开发经历，需要对技术有敬畏之心。总结为两点：

- 基础知识和理论知识非常重要，多多使用已有的且成熟的方案是关键。
- 对技术要有一颗严谨和敬畏的心，想清楚了再干，坚持高标准，很多事情都急不来。

2.4.2 薪资管理



对于薪资，我有一个基本的观点："对工作努力的人，要给予薪资方面的倾斜，即便他是亿万富翁，该多给的奖金一分都不能少，如果他经济有困难，还要更多给予倾斜。对不努力的人，无论多么困难，天平绝对不会向他倾斜。"公平、公正，是做事、做人的基本原则。

公司政策所导致的薪资不公平或不合理现象很常见，团队管理者的工作是让不公平现象在我们的管理下逐渐变得公平，这也是高效团队管理者的能力要素之一。很多时候，员工其实并不在意工资低一些，但是如果他们发现同级别的人工资比他们多很多，这时候就会出现强烈的内心冲突，一般这种情况你需要立即处理，安抚他们，并解释为什么会出现这种情况，并承诺后续如何平衡。

千万不要以为"公司内部不能谈论工资"这一条很有用，这条规定对于有大量同学在一起的大公司来说，几乎不起作用。我们要做的是，在公司开始慢慢有稳定的收入后，逐步调整薪资并向优秀的员工倾斜，让他们的付出得到合理的回报，并在薪资、奖金和期权分配上为他们争取更多的回报，这是对优秀人才最好的奖励。除了薪资以外，我们可以给予他们信任和空间，让他们承担更多职责，即使中间可能会搞砸，也要继续大力支持。其实，在成长的过程中，捅娄子的事情不可避免，但只要勇于面对、积极改进，一般不要在这些事情上过于苛刻。

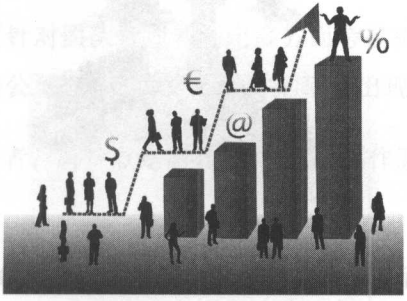
团队中其实不缺聪明能干的员工，但是限于种种制约，很多人并不知道该朝哪个方向去努力，这时技术管理者应该结合自己对业务、平台、团队等综合情况的了解和影响力，给大家指明一个方向，搭建好一个可以让他们施展的舞台，剩下的就不用太担心了，他们会做得很好。

总的来看，对于薪资，我们应该注意以下几点：

- 按照能力和他的付出进行衡量，不要把个人情感混淆其中。
- 制定标准化的考核规则，输出能够让人信服的流程和细则。
- 薪资的组成结构最好能够多元化，这样可以让员工在每一项奖金中感受自己的付出和回报。

- 薪资真的是第一要务，一定要把握标准，做到公平、公正，不然你的团队会垮的。

2.4.3　上升通道



基层程序员在工作几年后，一般有能力的都会被提升到 PL、PM、SE 等职位，员工也都想着能够被提拔，逐渐成为管理者。大家觉得，光做开发没有职业前途，永远都处在金字塔的底层。而在硅谷的公司，说话比较有分量、收入相对较高的人，他们有很多是在各层级中的技术佼佼者，他们备受尊重，干得也开心，不少人根本不愿意转做管理者。

编程其实是一门艺术，热爱和用心是非常重要的，相应的也容易出成绩。这就是为什么在计算机领域，如果做到顶尖程序员，一个人顶一百个很正常。如果程序员觉得没有前途，不思进取，而资质较好的程序员很快又被提拔为管理者，那我们的软件开发将很难有技术和人才的积累。

不要期望每个人都以同样的方式来进行管理。每个人在工作与管理时都有自己的风格与方法。重要的是关注结果，而不是方法。有一些事情你必须坚持，但这些往往都是基本的领导质量问题，而不是具体的管理方法问题。

我们作为技术团队管理者，应该多与每一位下属沟通，抽出时间来了解他们对于未来的期望，帮助他们逐渐设计符合他们想法的上升通道。当然，你给予他们机会，最终需要他们能够承担这个机会，承担机会意味着付出，如果这个人不愿意付出，只是口头不断表达自己的想法，那么所有的上升通道都不会向他敞开。

2.4.4 大规模变革

“没有什么比建立新秩序更困难、更无望成功、更危险的了。因为旧秩序的受益者都是改革者的反对者，只有那些可能从新秩序受益的人才会勉强支持改革。”

——Niccolo Machiavelli[⊖]

你一定要慎重执行大规模变革，无论是组织架构，还是人员任命，因为所有的改变，都会在人的内心引起权力欲望和恐惧。多做核心人物的思想沟通工作，不要由闭门小团体会议决定所有的改变方案，这样会让核心技术人员感觉到被边缘化、被决定命运。记住，只有多沟通才会让核心人员内心存在被尊重的感觉，而不是仅仅给他们加工资，人是有感情的，让他们感觉被尊重，胜过加薪。

2.4.5 一言堂

⊖ 尼科洛·马基雅维里，意大利政治思想家和历史学家，近代政治思想的主要奠基人之一。

技术团队如果出现喜欢搞一言堂的领导，逐渐会形成他的个人特权，刚开始不会让人感觉恶心，但是逐渐地他会越来越肆无忌惮，直到团队中有个性的、有能力的人全部离开，只剩下一些实在走不掉的、很能忍的，那么这个团队的整体战斗力就变得很弱。

总的来说，还需要针对团队管理者制定多方面考核机制，要有自下向上的反馈渠道，避免出现有违公司政策、违反劳动法的事情出现。

作为一名团队管理者，有一点是需要做到："不要作恶！"

2.4.6　频繁开会

根据 Business Insider 报道，美国员工每天总计需要参加 1100 万场会议，无效会议每年消耗美国公司 370 亿美元。苹果创始人乔布斯认为，会议规模越小越好，为此，他还曾拒绝过奥巴马邀请的一个科技大腕会议，原因是邀请名单太长了。乔布斯主持会议的风格就和苹果产品一样简单明了。他厌恶开大会，因为一个房间里有太多人的话就会争执不休。有一次，会议开始时他发现房间里出现了一个以往没有出席过会议的人（一位自称正在做市场营销项目需要参加此次会议的女士）。乔布斯询问对方是谁，最后礼貌地请她出去："我觉得我们不需要你在这个会议中。谢谢。"那位女士只好收拾好自己的东西离开了。

Alphabet（谷歌的全新母公司）CEO 佩奇曾经向全公司员工发送了一封题为

“如何高效开会”的电子邮件，邮件中的一条要点是：“每个会议都必须有一个决策者。如果没有决策者或者会议中无法产生决策，这个会就不应该开。”

我的一位同事，原先做人工智能算法，后来升为经理后，最多的时候一天参加 8 个会议，平均也有 4 个会议，你说他还能安心搞技术吗！现实工作中，由于组织复杂，中间层较多，各种各样的任务从上面下来，落实的方法就是各种各样的会议，所以现在很多研发员工的不少时间都被各种各样的规划、研讨、问题回溯、客户支持等会议占用。员工笑称：“白天是用来开会的，晚上加班才有时间编程序。”针对于不同的组织和项目，能尽快找出相应的沟通节点并能有效地减少这些沟通节点，是一个项目和部门领导需要经常思考的问题。

如果你喜欢有条理的议程，那么就让员工设定会议议程。最好能让员工提前向你报备议程。这样做，如果没有紧急的事，员工就有机会取消会议，也能让员工知道这是他的会议，占用的时间完全取决于他。因为是员工的会议，经理的发言应该只占 10% 的时间，剩下的 90% 都是倾听。

2.4.7 各种无端限制

如果你懂开发编程，一定知道什么是“锁”，锁就是用来同步和互斥。我发现很多开发部门里的各个开发团队间存在很多锁。比如：

- 技术能力上的锁。有一个项目需要在不同的地方做开发，这些模块用到不同的技术，比如 Java、C/C++、Python 等，但是这个团队里的每一个开发人员只懂一门语言，于是，需要配合，需要任务排期，同步互斥锁就很多。于是，一个本来只需要两个人干三周的工作变成了八个人干两个月。
- 负责模块上的锁。同理，不同的人负责不同的模块，于是一个项目需要集成好多模块，那么你就需要把这些模块的人找过来。和上面一样，每个人都有自己的时间安排，人越多，锁越多。于是，一个本来只需要两个人干两周的事，变成了八个人干一个多月。
- 时间锁、进度锁。有不同技能或是负责不同模块的开发人员有锁，有锁你

就要等，他们有自己的安排，所以，要协作起来，你就需要排期，去同步。而参与的人越多，你的锁就越多，你协调的时间就会更长。

- 沟通锁、利益锁。最恐怖的事情是，他们之间的沟通成本巨大。他们会花大量的时间来讨论一个功能是实现在你那边，还是他这边。每个人都有自己的利益和算盘，无形中增加了很多推诿、官僚和政治上的东西。

有时候，我们会觉得分工和分模块是产生效率的前提，但是实际情况并不是这样。我们可以看到，所谓的“分工”被彻彻底底滥用了，他们把“分工”当成了永远只干一件事的借口，一个程序员应该能够掌握多个语言，也能够负责多个模块甚至不同的职责。如果一个程序员觉得多学习一门语言，多掌握一个模块是件很困难的事，那么这个程序员本质上是不合格的。

2.4.8 老员工的管理

管理者遇到的一个最典型、最纠结的困难就是老员工处理问题。团队成长过程中必然有老员工和新员工，老员工甚至可能是你的朋友或者前同事。这种情况下，文化管理尤其重要，一旦管理不当就会出现很大问题，我们可以称这种情况为预期管理，预期如果没有管理好，有些人就会产生失落感，甚至对公司的方向和文化产生质疑。一旦一位很卓越的员工对公司文化产生质疑，后果将会很严重。

1. 避免生锈

一般情况下，初始团队都是金子，偶尔会有一些铁存在，在中国，铁可能就是老朋友，而土和锈一般不会出现，除非面试环节出错，但在初期也不容易被发现。团队中每个人的发展方向不一样，但有一个共同点，大家都希望朝着金子的方向走。

公司发展过程中，做的事情和范围必然扩大，当公司的成长速度和人的成长速度不匹配时，问题就会产生。如果人的成长速度慢于公司，那么公司要做的事和人能做的事之间就会产生一个缺口。这个缺口需要管理者去填补，可能需要加人或者其他人去做。但在填补这个缺口之前需要和老员工进行深入的沟通。你可

以让他去尝试做更多，去追赶公司的发展，但是如果能力上办不到，就要停下来做得少些，能力提升后再去做更多。怕就怕缺口已经产生，管理者却因为这个人是朋友或者老员工而忽略沟通，这时候老员工会产生失落感，而失落感将直接影响到一个人对公司的态度。

在公司成长很快、缺口很大的情况下，管理者必然要招新员工去填，这时候老员工心里的失落感就会影响他对公司的态度，当态度持续恶化时就会掉进锈里面。锈不会把金子拉下来，因为金子的愿景和舞台很大，但是铁会严重被锈影响，一旦铁被锈影响，更多的铁会掉进土里，公司的整个氛围就会变差。

所以在公司快速发展过程中，铁非常重要。大部分新的管理者不会去炒掉这个已经掉进锈里的“铁”，把他们当朋友，因为过往为公司做过很多贡献。但从一个理性的管理者角度来说，一旦发现锈在拉铁下来，应该马上采取行动。

2. 兑现承诺

在请有问题的人离开时，之前的承诺全部要兑现，这是一个管理者的道德。有很多管理者做不到，他们站在现在的角度看过去，觉得这个人与公司目前的要求差太多，顺理成章地否认他之前所有的贡献。作为老板要清楚，这个人把公司带到这一步已经付出了很多，他的付出值得之前所给的承诺。

一个小气的老板必然留不住人，即使在公司生意做得风生水起的时候，一旦发生不兑现承诺这种事情，员工也会受到很大影响，而公司里更多的铁会掉到锈里。这个公司可能在几年之内，或者几个月之内就会被毁掉，所以，请别人离开并不意味着你可以不遵守承诺。

处理掉进锈里的铁并不容易，执行过程中会存在很多困难，这里面会存在很多感性因素，比如这个老朋友有房贷、车贷，有老婆和孩子要养。但是一个公司的发展不是感性而是理性的，如果感性的决策做太多，就会有人开始不作为，其他人觉得他也可以不作为，因为他是老员工而那些拼命做事的人就会跳槽，因为在他们心里这个公司已经没救了。

2.4.9　工作时间

我的一位前同事，父母从家乡过来陪她一段时间，2 个月时间里没能抽出几天回家吃晚饭，她妈妈一怒之下拉她来辞职，因为在小城市生活的人来看，只有资本家才会这样压迫员工，哪有天天加班的道理。很遗憾，这就是我们这个行业的现状，不是某个人可以左右的。

谈到工作时间，其实大家想表达的是时间自由，即自己可以左右时间，而左右时间，很多人又会认为需要财务自由作为前提。这种想法不在少数，但是我认为，相比较财务自由而言，当代社会更多的是应该看是否能力自由。假设你现在有一笔钱，并且达到你所设想的财务自由标准，那你是否可以退休了？我想，真的让你退休了，你还会不自在，因为朋友圈逐渐没有了，紧跟着你会陷入更深的迷茫。所以，我认为，真正应该关注的是工作能力是否自由，如果你能力很强，又很善于掌握新的技能，那你完全可以自己掌控自己的时间，可以自己创业，或者去那些愿意给你时间自由的公司，一切事在人为，关键还是自身。

当然，不会有那么多具备这类能力的人，大多数的人还是需要上班的。而作为 IT 企业，还是需要体谅一下程序员的心理诉求。如果你可以做主，我建议针对程序员最好采用弹性工作制，多数企业中的多数雇员属于白天型的人，但与他们不同，多数程序员属于夜晚型的人。他们一般到了晚上才开始很有精神，对于开发工作也更加专注，所以如果有可能，尽量对他们使用弹性工作制，不要太过于限制工作的时间和地点。当然，很多开发工作是需要及时沟通的，所以每天必须有大于 4 个小时的时间让大家聚在一起，这样才能够完成需求、设计等评审流程，才能对具体的系统架构、代码问题进行讨论。此外，毕竟产品、市场、职能部门的工作人员不是弹性工作制，也要体谅且配合他们。

2.4.10　缺乏愿景

最有效的领导方法，包括两个能力：一是愿景（Vision），二是沟通能力。我们这里主要讲愿景。

什么是愿景？愿景就是作为一名软件工程师，未来你的发展前途、你做的产品对未来世界有什么影响等。说得直白一点，就是要让一名软件工程师对这家公司有归属感和对未来的期望，这些都是要能够落地的，给一个虚无缥缈（例如我们要做最好的科技公司）的目标，会让底层员工感觉不到真实。对一名软件工程师，你最好能够告诉他，你对公司很重要，但关键要说清楚为什么重要，特别需要让他们体会到他们的技术能够不断提高的愿景。

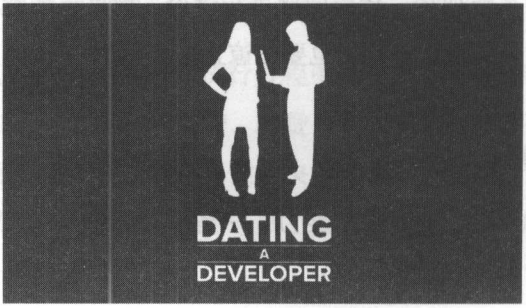
大公司有大公司的愿景，小公司有小公司的愿景，我们技术团队和个人应该也有自己的愿景。

团队里的个人可以有这样的愿景：三年之内成为 Web 前端方面的技术专家。团队可以有这样的愿景：打造国内技术领先的前端团队。

Jim Collins 的那本《 Build to Last 》，提到了所有 Visionary 公司都具有一个特质，就是很注重自身企业文化的建设。FaceBook 为了降低人才的流失风险，设立了相当严谨的内部人才建设规划，所有人都以成绩说话，只要完成一个重要项目并且在规定的时间内完成目标数据，公司立即对项目成员进行加薪和晋升。因此 FaceBook 打造了顶尖科技企业独一无二的 “黑客核心文化”，并发挥到了极致。

2.5 其他相关知识

2.5.1 理解程序员工作



技术为主的公司和非技术为主的公司，技术管理者所做的工作是有明显区别的，因此，对技术管理者的技术背景要求也是有明显差异的。非技术为主的公司，团队领导可以不是程序员出身，他可能来自业务部门，也可能来自运营部门，只要他过往的经历可以帮助他 Hold 住这个岗位。而对于技术为主的公司，技术管理者如果不懂技术，就无法和程序员进行交流，基本上参加会议时一句话也插不上，别人又怎么对你尊重呢。

Gace Hopper[⊖]在 1961 年写下了这些文字："程序员是一个古怪的群体，他们崛起的速度很快，很快就形成了独立的职业，并且过早地感染了不愿做出改变的抗性。我曾经听说有些程序员因为客户不愿意修改自己的系统而斥责客户，有时走进我的办公室，也会要求坚持他的想法。出于这个原因，我在办公室悬挂了一个逆时针走动的时钟。"

编程是一种有趣的工作，且大多数程序员都很享受工作，这样就不难理解了。为什么难以管理他们？如果有人付钱让你开心地玩，你还会愿意受制于人吗？受人管制就会减少工作中的乐趣！

程序员之间的差异非常大，只有很了解程序设计的人才能完全理解这一点。事实上，程序员之间的差异主要来自个人的内在因素，而不是外在属性。大多数公司的高层管理者对所有的程序员一视同仁，这种看法是片面的。微软公司的 Bill Gates、Adobe 公司的 John Warnock、FaceBook 公司的 Mark Zuckerberg 都没有犯这样的错误，因为他们也都是程序员。这也是为什么我觉得某些大型软件企业需要变革的原因，在科技界，你最好不要让非技术出身的人担任 CEO。

如果没有外界的干扰，许多程序员在独自面对自己的设备时通常都会很投入地写代码，一边写一边设计。技术团队管理者必须培养软件开发文化，而文化又是建立在可靠的开发实践基础上的，否则程序设计项目就可能失败。

成功管理程序员的关键是要认识到他们是独立的个体。程序员之间的差异很

⊖ 全球第一代程序员之一。

大，你必须努力地让每个人的长处都得到发挥，同时尽力提高或者至少抵消每个人的短板，这也是对技术管理者的要求。

因为程序员都是些无拘无束的人，常见的激励方法往往没什么用。除了进行必要的技术监督并把开发实践和过程落实到位之外，善于利用程序员的自我意识和改变世界的欲望也很关键。这就需要一类既能理解程序员的工作方式，又能理解工作本身的技术管理者，他们不仅能有效地激励程序员超常发挥，而且能按时交付结果。

此外，也不是只有计算机专业毕业的人才能做程序员。有一个同事，以前是学法律的，后来转行写代码，写出的代码比很多多年写代码的人还好。她在法律上的缜密思维，很好地转移到了代码逻辑上，不学自通，这就是程序员，你用正常思维理解不了他们的成长路线。

2.5.2 左脑型 VS 右脑型

右脑理论与左脑理论源于 Roger W.Sperry 的具体研究工作，根据他的研究表明，大脑的左半球和右半球具有针对不同任务的专门功能。左脑通常专用于分析任务和语言表达，而右脑主要用于空间感知任务、音乐等。左脑的表达能力比右脑强得多。

如果你是一名程序员，你很可能属于“左脑型”，这意味着语言、逻辑和分析使用得更多，也更客观。其实称为“左脑为主型”更恰当，因为我们只有一个大脑，两个半球始终是同时工作的。因此，你既可以是“左脑型”的人，又可以具有非语言交流、直觉、想象力较多且更主观等强烈的“右脑型”倾向，这些倾向通常更多地与音乐家、作家、艺术家等创新型人才相关联。

对于一名优秀的程序员来说，强大的左脑分析能力是必不可少的，不过，与右脑相关的活动往往也同样重要，这是因为程序设计是一门很有创意的艺术。事实上，我们发现，一些最顶尖的程序员同时也是音乐家。

《Thinking in C#》的作者 Larry O'Brien 和 Bruce Eckel 有这么一段评论："计算机程序设计非常有趣。与音乐一样，它也是先天才华与刻苦练习相结合的产物。与绘画一样，它也可以有多种发展方向，商业、艺术以及纯娱乐。众所周知，程序员的工作时间很长，但很少有人将其归因于富有创造力的狂热。程序员们在周末、假期甚至吃饭的时候讨论软件开发，不是因为他们缺乏想象力，而是因为其他人看不到他们想象的那个世界。"

2.5.3 程序员思维

程序员的思维有一个专业术语，叫作计算思维（Computational Thinking）。计算思维是根据计算机科学的基本概念和方法提出的，是用来理解需求、设计系统、实现编程、解决问题的思维方法。简而言之，计算思维就是描述程序员或计算机科学家是如何思考的。当然，计算机科学的理论知识如数理逻辑、离散数学、数据结构、算法以及面向对象等是计算思维的必要条件。计算思维有一系列的智力工具，不能一一尽述，仅介绍关键的几项。

1. 抽象思维（Abstract Thought）

给定一个问题，抽象就是去掉纷繁芜杂的与计算无关的部分，用规约（Reduction）的方法还原到问题的本质。所谓本质即把原来的问题转换为一个或几个可以使用计算机描述并解决的问题，进一步讲也就是转换成在算法上可计算的（Algorithmically Computable）一个或几个问题，更准确、更理论化、更上档次的描述是转换为邱奇 – 图灵论题（TChurch-Turing Thesis）可计算的可数个问题。图灵机（Turing Machine）和 λ 演算（Lambda Calculus）本身就是对可计算性（Computability）的漂亮的抽象，可以作为抽象思维的经典案例来学习。一般在实际工作中，常常需要把问题的实体对象根据需求表示为各种数据结构，如树、堆、栈等，而业务逻辑（Business Logic）过程表示为各种算法，如排序和查找等。

表示（Presentation）

表示是解决问题的第一步，也是关键的一步。在编程实践中，我们都有很深

的体会，一旦问题被准确地无歧义表示出来了，解决方案就烘云托月般地呈现出来了。这就是“数据即代码，代码即数据”的道理。抽象思维也广泛用于数学家的工作。面对一个困难的问题，数学家们常从两个方向开展研究。

一方面，从特殊情况入手，推广到更一般的情况；另一方面，将一个一般问题具体化成几种特殊情况。两个方向的结果最终汇聚在一起，就找到了问题的答案。我想这可能是论语中“我叩其两端而竭焉”的一个最好注解。而从特殊到一般就是一个不断抽象的过程。我们用一个具体的例子加以说明，有一个著名的“六度分隔理论（Six Degrees of Separation）”讲的是世界上任意两个人最多通过另外 6 个人就能相互认识，如果要验证这一理论，怎么做呢？

我们可以借助一个图来表示人与人之间的关系，每个人用图中的一个节点表示，如果 A 和 B 认识，那么在代表他们的节点之间有一条边连接。现在的问题就转换为检查这个图的直径是否大于 6。考虑到世界人口众多，且有生老病死，图的规模必然超大，并且是动态的不断变化的，算出它的直径仍需要更多的简化，这里就到此为止了。

2. 逻辑推理（Reasoning）

逻辑推理对于程序员的重要性不言而喻。与其说逻辑推理用于程序新功能的开发，不如说更多的应用在程序调试和修改 BUG 的过程中。程序调试有点类似于 Sherlock Holmes 侦破案件的过程。和 Dr. Wason 比较起来，Holmes 的推理优于常人的地方有两点：第一，在观察现场或听取来访者叙述时，他能够得到更多的数据，尤其是一些别人容易忽略的关键细节，这得益于他对犯罪领域知识的丰富积累，知道什么才是更重要的数据；第二，根据得到的数据，他能够联想到更多的可能性结论，这得益于他大量的案例储存。有了这两点，就能够通过一环套一环的推理链逐渐缩小侦察范围，最终认清犯罪事实。

程序调试也是如此，首先必须掌握程序执行过程的细节。然后从问题出发，分别朝着产生的原因和导致的后果前后两个方向推理。逐渐定位问题的范围，最终找到问题的根源和解决的方案。我们比 Sherlock Holmes 幸运的是，可以借助

调试工具来了解程序运行的过程。所以，一个不能使用调试工具的程序会令程序员感到无比沮丧，只能通过跟踪信息来跟踪程序运行的过程。如果不知道程序运行的过程，推理就只能靠猜，那么修改Bug是非常危险的，很容易导致回退（Regression）的错误，这种情况下如同盲人摸象，根本不知道自己在做什么。另外，Sherlock Holmes还多次表达过这样的观点，案子越是离奇，越容易解决，因为奇怪的点都是线索。

对程序员来讲，也不必担心奇怪的问题，奇怪本身就是线索。关键看对程序运行细节的了解程度和逻辑推理的技术水平。

3. 分析（Analysis）

分析是上文提到的数学家所用思维方式中从一般到若干特殊情况的过程。面对一个问题，如果一下子描述不清楚或者表示不出来，可以先找出满足问题条件的几种特殊情况。通过仔细检查这几种特殊情况，求同存异，找出他们共同的规律或模式，并对这些模式或规律加以验证，就可以找出描述或表示问题的方法。这就是猜测加验证（guess-and-verify）的过程。项目需求分析时常见的应用案例分析（Use Case Analysis）方法，就是用一个个具体的使用案例将模糊的项目需求生动地表达出来。

4. 分解（Decomposing）

把一个大问题分解为几个小问题，或者把一个复杂的过程分解为几个子过程，这样有助于问题的解决。这也是程序员常用的手段，如算法策略中的分而治之（Divide-and-Conquer）和合并排序就是这方面的例子。

5. 递归（Recursion）

对于初学编程的人，递归可能是一个比较诡异的、较难掌握的概念。但是一个程序员如果不懂递归，很难称之为程序员。因为很多稍微复杂的算法他都不可能理解，如回溯和动态规划，甚至于树的遍历。递归常常可以用简单的方法非常优雅地表达复杂的算法。

另外，有关计算思维的特有方法还有并行、异步 / 同步、模拟 / 近似、优化、分层、封装、解耦等。程序员的思维艺术即计算思维不是一两天的短时间内可以形成的，需要在实践中慢慢琢磨，不断提升，且永无止境。

2.5.4 杰出的程序员

杰出的程序员是如何产生的呢？仅仅具备程序设计方面的天赋是远远不够的。杰出的程序员是大师级的人物，做事有条不紊、遵守纪律，能够凭借直觉把代码和程序组织好，能够约束自己总是在编写代码之前进行设计，能够在最少的时间内编写出清晰、简洁、实用、高质量的代码并获得预期的结果。换言之，杰出的程序员是大师级的工匠。

如果程序员的学习、工作动力主要来源于项目管理时间表、管理层的压力，或者金钱，那么他不会成为一名杰出的程序员。对大多数杰出的程序员来说，动力实际上来源于更高的追求，例如改变世界，做出人们喜欢使用的程序或产品。杰出的程序员希望并且需要为具有世界影响的项目而工作，他们希望能够感受到自己的工作是有意义的，哪怕只在某个很小的方面有意义也行。杰出的程序员偏爱能够满足他们更高理想或要求的公司和项目，他们非常在意自己所做的事情，常常为了想要的结果而超负荷工作，而不会在某种压力下自愿做低技术含量的重复劳动。

美国科罗拉多大学早在 1993 年就做过一个关于软件工程师的研究，报告显示：“和普通工程师相比，那些出类拔萃的工程师往往更能照顾全局，更喜欢实际行动，更易受使命感推动，更能展示和表达出一种坚定的信念，在管理中更容易发挥主动的作用，更能帮助其他工程师。”

世界上杰出的程序员不多，不可能让每个项目团队都拥有杰出的程序员。而且多数团队也只能容忍队伍中有一两名杰出的程序员。大多数的程序都需要靠普通程序员完成，他们通常是称职的、专业的、能干的，但是可能会把程序设计看作为一种工作，而不是追求。

2.5.5　小型团队 VS 大型团队

一般来说，小型团队的效率比较高。一个小型团队可以就近办公，经常可以聚在同一个办公室，即便是敏捷团队，规模可能更大，但是也提倡就近办公的原则。

高效的小型团队会在所有成员之间平等地共享所有沟通信息，坚持做到遇到问题时能够立即回答，当发生设计难题时马上解决，并互相提供调试相关的建议与帮助。遇到问题寻求人帮助的时候，不需要浪费时间在流程步骤与消息的往来循环上。

随着团队人数的增加，沟通也会变得更加困难，沟通交互变得碎片化，这样会导致更多的错误预设，以及其他在小型团队中能够避免的错误步骤。

对于大型团队来说，可能需要从外部招聘团队管理者，这时候你需要了解他的管理风格，并成功交付项目的实际经验。如果可能的话，找到曾经与他一起工作的人，看看他们是如何看待他在这两个方面的表现的。LinkedIn 的出现大大简化了寻找证明人的步骤，要充分利用它。最好的方式是从内部培养团队管理者，一般来说，能够变成杰出管理者的优秀程序员很少。对于那些有才能，并且期望能够成为高效管理人员的程序员，我们应当鼓励、培养和奖励他们。随着组织的发展，他们可以产生显著的积极影响。

大公司可能会采用“矩阵管理”方式组建团队，这意味着团队成员都有各自隶属的职能领域，但为了完成某个指定的项目，被“临时”分配到了一个矩阵型的团队中。这样能够在项目人事配备方面获得最大的灵活性，但是也带来了考核的难题。如果团队成员很优秀，这不会是一个问题。但是当团队成员表现不佳时，则可能成为“临时”团队负责人的烦心事。这个问题通常被称为有责任但没有权力。

2.5.6　扁平化管理

雷军曾经在一次采访中谈到小米的情况：“小米团队是小米成功的核心原因。

和一群聪明人一起共事，为了挖到聪明人可以不惜一切代价。如果一个同事不够优秀，很可能不但不能有效帮助整个团队，反而有可能影响到整个团队的工作效率。真正到小米来的人，都是真正干活的人，他想做成一件事情，所以非常有热情。来到小米工作的人聪明、技术一流、有战斗力，这样的员工做出来的产品注定是一流的。”

小米的组织架构没有层级，基本上是三级：七个核心创始人 – 部门 leader– 员工。同时不会让团队太大，稍微大一点就拆分成小团队。从小米的办公布局就能看出这种组织结构：一层产品、一层营销、一层硬件、一层电商，每层由一名创始人坐镇，能一竿子插到底的执行。大家互不干涉，都希望能够在各自分管的领域给力，一起把这个事情做好。

从小米的团队管理，我们已经可以了解到扁平化的好处，它更容易让能干的人冒出来。理论上来说，通过减少管理层次、压缩职能部门和机构、裁减人员，使企业的决策层和操作层之间的中间管理层级尽可能减少，以便使企业快速地将决策权延至企业生产、营销的最前线，从而为提高企业效率建立富有弹性的新型管理模式。扁平化管理解决了传统的金字塔状的企业管理模式的诸多难题和矛盾，是解决层级式管理在现代环境下面临的难题而实施的一种管理模式。当企业规模扩大时，原来的有效办法是增加管理层次，而现在的有效办法是增加管理幅度。当管理层次减少而管理幅度增加时，金字塔状的组织形式就被“压缩”成扁平状的组织形式。

为什么过去没有这么强烈的诉求？因为中国的 IT 起步较晚，人才的出现、累计需要很多年，现在已经出现了大规模的知识型员工。知识型员工的一个鲜明特点就是，对专业的忠诚大于对所服务的企业的忠诚，选择企业的目的是致力于寻求能够实现自身专业成就最大化的成长平台。200 多年的工业社会发展使专业分工模式越来越成熟，也进一步催生了庞大的知识型员工群体。知识型员工群体的兴起和“去中心化”的信息传播方式，这两者结合，对企业管理提出新的要求，同时，这部分群体对企业的绩效发挥着越来越大的影响，而知识型员工又是社群自组织和传统组织混合协作的主要群体。

2.5.7 金州勇士奇迹

在2015～2016年的NBA（美国职业篮球联赛）赛季，位于硅谷地区的金州勇士队（Golden State Warriors）创造了NBA历史上常规赛获胜率最高的纪录，在全部82场比赛中获胜73场。而在一年前，该队获得了NBA总冠军。

但事实上，勇士队长期以来一直是NBA里的一支“鱼腩球队”。在2009年，金州勇士队还是NBA里最烂的球队之一，那一年它的成绩排名倒数第二，当然勇士队也不可能有大牌球星和教练。因此该队能取得这样的成绩，实在是一个奇迹，而它创造奇迹的方式在体育史上恐怕是独一无二的。

金州勇士队的成功并非砸钱的结果，而是因为它处在一个特别的地区——硅谷。

硅谷地区有两种人最不缺，即风险投资人和工程师，勇士队的奇迹从很大程度上讲是靠他们创造的。前者善于看到其他人还没有发现的投资潜力，然后把它经营成值钱的实业；后者善于利用技术创造奇迹。

勇士队的成功就是他们合作的成果。6年前勇士队的比赛成绩跌到了谷底，因此价值较低，一些风险投资人决定将这支不值钱的球队买下来好好经营，让它成为美国体育界最耀眼的明星。

这个计划看上去有点疯狂，不过投资人有自己的考虑，他们有秘密武器，那就是应用大数据的工程师。最终，投资人用4.5亿美元这个相对较低的价格完成了对勇士队的收购。

在收购完成后，投资人为球队委派了新的管理层：没有任何执教NBA经验的史蒂夫·科尔，因突出的投篮优势被委任为教练。科尔在执掌勇士队之后，坚持用数据说话，而不是凭经验。

他根据背后团队对历年来NBA比赛的统计，发现最有效的进攻是眼花缭乱的传球和准确的投篮，而不是彰显个人能力的突破和扣篮。在这个思想的指导下，勇士队队员苦练神投技，全队在一个赛季中投进1 000个三分球，又创造了

一项 NBA 纪录。

这其中，最亮眼的新打法是尽可能地从 24 英尺（大约 7.3 米）外的三分线投篮，这样可以得 3 分。正是因为不再按照篮球传统的战术作战，勇士队卖掉了那些价钱高却效率低的明星，而着重培养自己看中的新人。

这位新人叫斯蒂芬·库里（Stephen Curry），三分球的神投手。在 2014 ～ 2015 赛季中，库里的神投让勇士队夺得了 40 多年来的第一个总冠军，他自己也成为当年的最有价值球员（MVP）。到了 2015 ～ 2016 赛季，库里投进了 403 个三分球，创造了 NBA 历史上的纪录，打破了由雷·阿伦所保持的个人单赛季 269 记三分命中数的纪录。

除了利用数据制定战略，勇士队还利用实时数据及时调整比赛中的战术。早在 2012 年，勇士队的总裁兼 COO（首席运营官）里克·威尔茨（Rick Weltz）就在一次大数据会议（TUCON 2012）上介绍了该球队应用大数据的成果。根据威尔茨的介绍，大数据可以帮助球队改进精细到两个人配合的细节。正是靠高科技，勇士队才得以在短短 6 年里从倒数第二名登顶 NBA 的总冠军。

鉴于勇士队的战术和成绩给 NBA 带来的巨大冲击，奥巴马在白宫专门接见了勇士队，并且讲道："（这）看起来正在打破这项运动的格局，这似乎是不公平的比赛。"篮球界的人士则认为，勇士队是 NBA 里的 Google。

2.5.8 KPI 之祸

索尼公司前常务董事天外伺朗的《绩效主义毁了索尼》一文，曾经在业界流传甚广，也激起了广泛的争议，支持的、反对的意见和声音到现在都还没有停止。抛开索尼是否真正理解 KPI 等争议，单纯从文章描述的现象来看，相信绝大部分公司里面都会存在类似的现象，例如：

- 因为要考核业绩，几乎所有人都提出相对容易实现的低目标。
- 因实行绩效主义，索尼公司内追求眼前利益的风气蔓延。这样一来，短期

内难见效益的工作，比如产品质量检验以及“老化处理”工序都受到轻视。

- 上司不把部下当有感情的人看待，而是一切都看指标。
- 为衡量业绩，首先必须把各种工作要素量化。但是工作是无法简单量化的。公司为统计业绩，花费了大量的精力和时间，而在真正的工作上却敷衍了事，出现了本末倒置的倾向。

大多数人开始带技术团队后，在绩效考核这方面同样遇到了类似的疑惑，例如：

（1）程序员的工作怎么量化？Bug 数？代码行？版本数？

做过程序员的都知道，这些指标都是不可行的。假设公司考核程序员的 Bug 数和等级，并且同时也考核测试人员发现 Bug 的数量，结果程序员和测试员为了一个问题是 Bug 还是需求遗漏、Bug 等级是严重还是一般，能够吵上 2 个小时，于是最后就看谁会吵，谁官大，搞得程序员和测试员身心俱疲，关系很紧张！

（2）即使程序员的工作可以量化，每次绩效都是这几个指标，定绩效目标还有意义么？

例如，假设考核程序员用 Bug 数、代码行数、版本数，2000 年用这个指标，2017 年也还是这个指标，这样的绩效目标有什么意义呢？

（3）团队 Leader 如何制定团队的 KPI？

例如，可以看两个团队谁的代码行多么？可以看谁的团队 Bug 数多么？可以看谁的团队版本数多么？可以看谁的团队分享次数多么？这些其实都不行。

（4）前瞻性的工作谁愿意做，有风险的工作谁愿意做？

例如，引入 ElasticSearch 理论上是可以提升搜索性能的，但可能在引入的这一年反而会带来很多问题，而能带来多少收益还不确定，这个时候怎么定 KPI？这个时候推荐使用 OKR。

OKR 全称是 Objectives and Key Results，而 KPI 的全称是 Key Performance Indicators，OKR 和 KPI 具体的差别表现在：OKR 的关键词是 Objectives，KPI 的关键词是 Indicators！

不要小看了这两个词的力量，正是这两个词决定了 OKR 和 KPI 的本质差异：OKR 关注的是目标，KPI 关注的是指标。当我们关注“目标”的时候，我们会思考接下来我要做的事情是什么；而我们关注“指标”的时候，我们会思考自己的工作如何评价。

- 以程序员为例，如果我们关注目标，我们会想接下来我应该做什么事情，是要解决产品的卡顿问题，还是引入大数据来做精准推荐；如果关注指标，因为我们的工作是编程，那我们就会想哪些指标可以衡量编程工作呢？我们想到的是代码行数、Bug 数、单元测试覆盖率等。
- 以足球运动员为例，如果关注目标，我们会想到夺冠、四强、保级；如果关注指标，那我们就会想到进球数、助攻数、跑动距离、比赛场次等。
- 以滴滴和快的为例，如果关注目标，快的的目标应该是超越滴滴；如果关注指标，快的的指标应该是司机数量、订单数、乘客数等。

为何这两种思考方式差异如此大呢？有一句名言形象地说明了这一点：如果方向对了，就不怕路途遥远！如果方向不对，指标再漂亮都没有意义，甚至指标越漂亮就错得越离谱。目标是我们的方向，指标是评价我们做事情的质量。使用 OKR 的时候，我们的第一反应是：“我们的目标是什么？”而使用 KPI 的时候，我们的第一反应是：“我们的职责是什么？”如果我们将思维固化在当前的职责，那就不会去审视整个环境当前的状态以及后续可能发生的变化，也就不会及时地根据实际情况进行调整。

彼得·德鲁克在《管理的实践》中说：“并不是有了工作才有目标，而是相反，有了目标才能确定每个人的工作。所以企业的使命和任务，必须转化为目标。”我觉得这句话非常好地诠释了 OKR 的本质，以及 OKR 和 KPI 的区别，形象地提炼一下：OKR 让我们做正确的事情，KPI 让我们正确地做事情！

OKR 目前在美国硅谷的科技公司应用并取得了很好的效果，但介绍 OKR 的文章里面无一例外都提到了 OKR 和绩效考核无关，例如 Facebook 的绩效考核是 360 度环评，而中国公司的绩效目前来看不太可能采用这种方式进行绩效评价，

如果我们要推行 OKR，绩效考核如何做？难道还要发明另外一套机制来进行考核？

前面我们分析 OKR 和 KPI 的关系的时候，提到了 OKR 其实可以用 KPI 或者 Milestone 的形式来进行衡量，而这正好和我们传统的 KPI 绩效考核的形式是一致的，因此我认为根据 OKR 来进行绩效考核并没有什么问题，而且可以从已有的 KPI 绩效考核平滑地过渡到 OKR 绩效考核，只要考核 OKR 的 KR 的达成情况就可以了。

2.5.9 团建活动的技巧

很多团队为了加强成员之间的了解和协作，并建立成员对团队的认同感，每年都会组织 Outing 活动来进行团队建设，但我们看到越来越多的情况是 Outing 变成了单纯的休闲旅游，并且越来越高端，现在已不满足国内游了，很多团队开始出国游。当然，如果这是公司的一种变相福利无可厚非，但是是否真的很有帮助呢？尤其是人数超过十个以上，大家出去玩，会各自找自己相熟的人一起玩，最多说说一些无关痛痒的话，Outing 结束了一切照旧。

个人比较倾向于户外活动，最好是有一定强度并过夜的活动，当然最好能够找专业的户外拓展公司来协助，效果会更好。在荒无人烟的野外，通过刻意划分的小组、强制的行为约束、集体的协作，共同挑战艰苦的旅程，在别无选择和共同的困难面前，人和人的心会迅速打开和贴近，自觉地相互帮助共同完成目标，再通过晚上的喝酒助兴进一步加强和巩固团队关系。

户外活动过程中，会遇到很多困难和挑战，每个人都需要团队的鼓励和帮助才能成功突破自我，这特别能体现团队协作力量的伟大，这和我们工作中遇到的问题解决思路是一样的。

3

第 3 章

产品开发过程管理

理论好比是教人如何在地图上规划一条路，而实践则好比是要在现实中一步一个脚印地走完这条路。虽然从地图上也能看到山川河流的标记，但是和实际的跋山涉水、翻山越岭相比，还是有天壤之别。不仅地面上的荆棘丛生在地图上看不到，而且突然从山上滑落的巨石更是制定计划时意料不到的。理论和实践之间的这个距离，是成为一名成熟的开发经理[⊖]所必须跨越也是最难跨越的鸿沟。

工程延期、Bug 丛生、沟通冗长、人心浮动，这些现象可能是因为开发经理个人管理能力或经验不足，也可能是整个细分行业不够成熟，没有形成方法论，造成各种混乱。其实这种现象在全球 IT 行业都是普遍存在的，需要大家一起透彻地总结、分析、实践。

软件工程是一门综合学科，因此软件工程师的培养本身就需要大量的计算机语言和计算机知识，还有相关的数学、物理、电子电路等知识，门槛很高。而想要对软件和计算机之外的行业业务实现模拟，软件工程师还要对该业务所在行业的专业知识进行一定的积累，并要跨越听得懂和能执行两个阶段，然后才能够用

⊖ 我们这里假设开发经理就是产品开发经理，本章都用开发经理代称团队管理者。

另外一种语言，也就是计算机语言表达出来。这是一个相当高的要求。

本章主要介绍和解决以下问题，这些也是全书的基础：

- 开发经理的工作内容和个人品质。
- 产品开发过程全管理细节。
- 产品开发过程管理需要注意的事项和细节。

3.1 开发经理及研发体系介绍

无论是传统企业，还是互联网企业，软件的开发生命周期基本上都包含以下几个关键的活动：

- 确定业务需求
- 编码
- 测试
- 上线

这些活动按顺序发生，每一个活动都是下一个活动启动的前提，而且必须严格按照时间顺序推进，才能最终得到结果，这是一个典型的生命周期过程。不同的活动，从项目启动开始，随着时间的积累，直到软件上线结束。当业务发生了变化之后，如果这个软件需要修改，会再次经历一个完整的开发生命周期。和上一个开发生命周期的区别在于，本次生命周期的起点是上一次开发生命周期的终点，也就是说，每一个开发生命周期本身也是按照时间顺序排列的，这些开发生命周期组成了软件的生命周期。

理解了软件的生命周期后，管理人员要学会用科学的工作方法来管理软件开发生命周期，而不是僵化地照搬流程。流程的作用是规范开发过程，你必须理解后才能使用。

绝大多数 IT 公司，无论是软件外包公司还是产品研发公司，都会按照行业划分事业部，而项目组是事业部最基本的业务运作单位，各事业部内设专职的开发经理，开发经理对项目（产品开发）的全过程负责，因此他是公司最重要的基层管理角色之一。大多数公司的产品开发过程又会作为一个项目来进行运作和管理，所以我们这里就按照开发经理制，讲讲开发经理（产品开发经理）的职责及其所需具备的品质。

3.1.1 开胃故事

记得读书的时候听过一个故事，说某市公务员面试题是这样的：

请你对以下4件事情进行排序，说清楚具体应该按照怎么样的顺序执行。

（1）有一位你的老上级到北京疗养，明天约好了见面；

（2）明天上午10：00需要会见重要的外商；

（3）下属今天送到了医院，正在动手术；

（4）明天需要完成部门年度工作总结。

我们不去关心这个考题的真伪，当时报纸上说的正确的处理方式是：

（1）早上6点起床，7点半到达老上级住所，陪同吃早饭；

（2）吃早饭时向老上级汇报今天需要会见重要外商，请老上级谅解，不能多陪；

（3）老上级一定支持你的工作，继而转去会场；

（4）去会场途中打电话，安排助手去医院看望手术后的下属并慰问家属；

（5）会见外商结束后，立即赶回单位，组织人手编写年度总结，并通知可能晚上需要加班。

3.1.2 什么是软件

1. 软件发展历史

从冯·诺依曼结构开始，程序逻辑开始脱离硬件，采用二进制编码。软硬件两者结合，一个可编程的大脑就出现了，这也是为什么我们把计算机叫作电脑的原因。在硬件上运行的程序就是软件，用来控制硬件的行为。通过编写软件，可以制造出各种各样具备不同能力的“人”，而这所花费的时间会比培训一个真正的人少很多。

随着半导体技术的进步，硬件的成本越来越低，但性能越来越高。摩尔定律：当价格不变时，集成电路上可容纳的元器件数目，约每隔 18 ～ 24 个月增加一倍，性能提升一倍。软件方面，为了简化难度，开始采用汇编语言，进而出现了类似于人类语言的高级语言，比如 C/C++/Java 等，这使得人类可以用这些高级语言，把人类的知识传递给计算机，并训练计算机掌握某种技能。

软件的发展历史，可以说是用机器模拟人的进一步发展。在计算机出现前，人们用机器来代替人进行生产，也就是所谓的机械化生产。软件的出现，也许很多人没有意识到，包括这个历史过程中的参与者，实际上是人类有意无意地在计算机上模拟自己这种原始动机的体现。人们对于时间的恐惧，导致人们不断地去为延长自身生命而努力，提升自己的生产力是其中的一种途径。软件可以让人们节省大量的工作时间，从而有更充分的时间去关注并推进自身的核心生命周期。

软件模拟出了一个虚拟的世界，人们在自己创造的虚拟世界中互相联系，形成了一个虚拟的社会，从而给人类带来了极大的便利。换句话说，软件是人类实际需求的虚拟化实现，通过软件让机器完成本应由人类完成的工作，这样人类可以做更多有创造性的事情，可以更多地休息、享乐。

2. 软件的作用

软件的主要目的就是把人类生活的非核心生命周期软件化、虚拟化，以提供更低的成本和更高效率的新生活，让核心生命周期的运行能够更加容易，让非核心生命周期的处理更少地占用人类的时间，变相地延长人类生命。这正是人工智能的可怕之处，因为人工智能强大之后会存在替代人类的核心生命周期的可能。

企业在采用软件之前，需要先搞清楚软件可以帮助企业做什么。软件的出现，确实可以降低业务的成本。但是在没有软件的情况下，业务也是一样能跑的。如果只是为了跟风而采用软件，说不定反而提高了企业的经营成本。想想企业引入的 ERP 系统，真正能够帮助企业的成功案例有多少？我们经常说软件或技术是业务的使能者（Enabler），实际就是把业务的成本从很高降到了很低的程度而已，并不是有了什么新的业务。另外，软件也不是降低业务成本的唯一方式。

3. 软件生命周期

软件的生命周期可以拆分为软件开发生命周期和软件运行生命周期。软件的开发生命周期结束后，生成的是软件，紧接着软件运行生命周期就开始了。一个完整的软件开发生命周期推进的过程，一般称为软件的研发。执行软件开发生命周期的人员，称为软件研发人员。参与一个软件的所有研发的人员的集合，一般称为软件研发团队。

软件的出现使得机器的生产生命周期产生了分工，形成了硬件生产生命周期和软件生产生命周期两种形式。

人们对世界的访问生命周期发生了切分，形成了软件的访问生命周期，从而打破了物理时空的限制。对时空限制的突破，极大地提高了生产力，人们不再需要花费大量精力来跨越时空的限制，只需很短的时间就能够做到更多的事情，从而完成更多的生命周期。

3.1.3 什么是开发经理

对很多从事技术工作的朋友来说，开发经理不仅是个“光环笼罩”的职位，也是走向管理的一条“捷径”。但是，“捷径”并不代表最快，更不代表很容易。开发经理更像是个专业岗位，需要依靠领导力来解决和激励团队，靠沟通和协调推进项目，甚至需要一定的政治和文化意识才能获得支持。这些软能力很难通过理论教导获取，更多的是要通过实践、经验分享方式获取。

写在前面：无论多难，都要记住一点，只要别人不赶你走，你就厚着脸皮待下去，这样你才有可能熬到项目成功。

开发经理职务描述

开发经理是公司委派的负责实现产品开发目标的个人，是公司授权的产品开发负责人，是产品开发的直接组织者和领导者。

开发经理的职责

1）对产品开发全过程进行组织和管理，按预期交付产品开发的成果；

2）管理产品开发团队，为成员提供技术和业务上的支撑，使之高效而愉快地工作，并获得最满意的工作体验；

3）管理对外关系，以取得外部对交付成果及过程的最满意评价。

也就是说，一个合格的开发经理必须同时做到“按预期交付成果”“让客户满意”“让员工满意”。

开发经理的主要任务

1）与产品经理交流。IT 项目 / 产品一般比较复杂，实际交付或上线风险较大，需要在项目启动前约定好工作范围、进度计划，要估算成本和人力资源。开发经理需要近距离地了解需求、资源等约束，这样制定的项目实施方案才会切实可行。与产品经理交流不仅有助于开发经理深入了解客户需求，也可以帮助产品经理了解开发经理的能力。

2）负责产品开发交付。明确产品开发立项之后，开发经理负责围绕预期目标，遵循确定的规范执行项目。开发经理不仅要制定思路清晰、考虑周密的计划，还要调整资源、委派任务，推进计划的执行。执行过程中，还要及时处理出现的问题，定期向相关人员汇报进展，保证在规定的时间和预算内交付项目进展。

3）完成产品开发收尾。完成交付成果之后，要将成果移交相关部门，确保相关部门可以稳定地使用系统。然后，将后续服务移交给服务部门，确保客户得到后续的服务保障。开发经理交付的成果直接决定客户满意度，影响客户是否愿

意付款，因此公司还可能会要求开发经理（如果同时身兼项目经理的情况下）配合交付部门完成收款工作。

4）管理干系人的关系。一个IT项目可能涉及投资方、客户、分包单位、合作伙伴，甚至可能包括政府、社会等各方面的关系，面对的客户也不只是一个人，而是一个群体。开发经理作为各方的桥梁和纽带，要随时处理各方信息，保持密切沟通，解决矛盾冲突，只有这样才能让客户满意。

5）管理项目团队。由于项目团队的临时性（有可能开发经理并不是团队的实际负责人，只是普通一员，在该项目中被选为开发经理，职级上和团队成员平级），开发经理需要花费很大的精力寻找合适的资源，优化资源配置，建立合理的组织结构，确定清晰的职责分工。项目过程中还需要通过各种措施进行团队建设，打造高效团队。

从这些工作任务的性质来看，开发经理是项目的推动者、技术的输出者，也是关系的协调者。总的来说，开发经理往往是决定一个项目成败的关键人物，要求其职业素养高、技术能力强、综合能力强、职责范围广。也正是因为要求太高，所以很多公司将开发经理、项目经理区分开，即项目经理可以不用管技术，专心做协调、进度把控工作。

题外话：关于项目经理职位，想要走纯项目经理路线的同仁注意了，这是一条无悔路，后面会遇到的困难无数，如果半路发现实际情况和最初的想象有很多差距，觉得自己不适合做项目经理，那问题可就大了。因为，这不仅对外部部门和团队会造成非常大的不利影响，对自己也非常不利，毕竟技术的发展实在是太快了，再想退回到技术路线的难度很大。我们这里还是主要讲开发经理。

3.1.4　开发经理素质

作为开发经理，有几个基本素质是必须具备且非常重要的：

1）领导力。领导力是指通过领导他人来完成工作的能力。开发经理虽然是项目领导核心，但需要依赖团队完成任务。由于项目组的动态性和临时性，在一些公司，开发经理对于团队成员并不具备完全的管理权力，所以更需要将一组成

员凝聚成一个团队，激发和影响他人为了共同的目标而努力工作。开发经理和大家是平等的，很多时候还要走在别人的前面。你不仅要自己先做到要求别人做到的事，而且要知道“下一步”应该干什么，下一个目标在哪里。开发经理的口号应该是“跟我冲”，而不是“给我上”。

2）责任心。项目执行过程中会遇到很多困难，经常会超出预期。这个时候，能够帮助开发经理挺下去的可能只有“责任心”了。具备强烈责任心的人，出于对承诺的负责，会倾尽全力达成目标而不言放弃，因此也是可靠、可信的人。这样的人，公司、客户和团队才会对其放心，才会全力支持。具有强烈责任心的人，也会非常注重细节，能主动发现问题，会时不时地在脑子中模拟一件事的执行过程，设想各种意外情况，考虑如何应对。这样的人比别人看得远，看得透。

3）积极主动。积极主动的人最大的特点是不会经常抱怨，因为抱怨本身是没有用的。开发经理是项目组的“主心骨”，如果这个人很容易慌乱，那这个项目也会混乱。开发经理必须时刻保持好的心态，如果团队成员始终看到一个信心满满、镇定自若的开发经理，大家也会充满信心。如果团队总是看到一个整天愁眉苦脸、满腹牢骚的开发经理，大家可能担心他随时崩溃，自然自己也不会积极主动了。

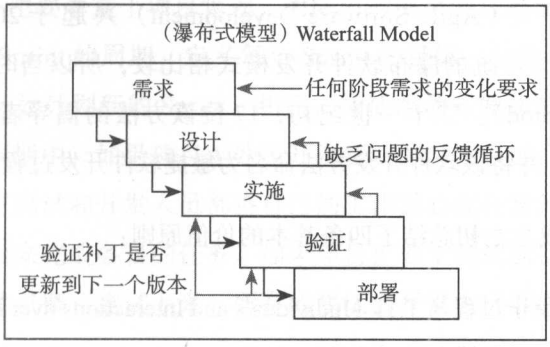
4）抗压能力。项目中会出现各种突发事件，有时候需要忍受极大的压力。有抗压能力的人在困难来临的时候仍能镇定自若，仍能冷静思考，即使在无能为力的时候，也能保持“风度”和“幽默感”，从而稳定军心、解决问题。很多技术出身的经理其实不太能承受得住压力，一旦他们被领导骂了或者被施加了压力，他们会直接把火向下传导，不知道在自己这一层控制一下，先调查清楚再发火（最好是不要发火，而用沟通的方式解决问题），这是他们管理团队的一个弱点，需要不断改进，否则会出现团队成员激烈反抗的情况。

3.1.5 软件开发模式介绍

1. 瀑布式开发模型

瀑布式开发模型是一种严格按照需求—设计—实施—交付四个阶段进行软件开发的模型，并且在各个阶段结束时要经过严格的评审，只有当能够确认一个阶段的开发成果是正确时才能够进行下一阶段的开发。

瀑布模型的四个阶段中，除了分别完成本阶段所定义的活动之外，都必须进行项目管理、质量保证、配置管理和测试活动，这四个活动的过程贯穿整个瀑布型软件生命周期。



瀑布式开发模式本身是没有问题的。比如在建筑行业，其实就是瀑布式开发模式，在需求做完后进行设计，在设计完成之后，再开工建设。区别在于，建筑行业是对空间切分，而空间是现成的，只需要顺从地球的自然地理环境以及地球的重力等因素，大部分都是可预测、稳定的。建筑设计师本身也是建筑的使用者，对用户的需求感同身受，容易理解。

但软件行业存在不同，软件是用来代替业务人员的，首先需要理解业务人员的工作，才能做出设计。而理解业务人员的工作会存在各种各样的困难，特别是沟通的偏差会导致软件的起点会有问题。软件工程师本身的门槛很高，如果没有一位强有力的技术管理者牵头，很容易出现设计阶段普通程序员闲置的情况，因此软件行业采用瀑布式模型时，会存在大量的浪费，最终的结果也往往不尽如人意，这也是为什么其他开发模式兴起的原因。

2. 敏捷开发模型

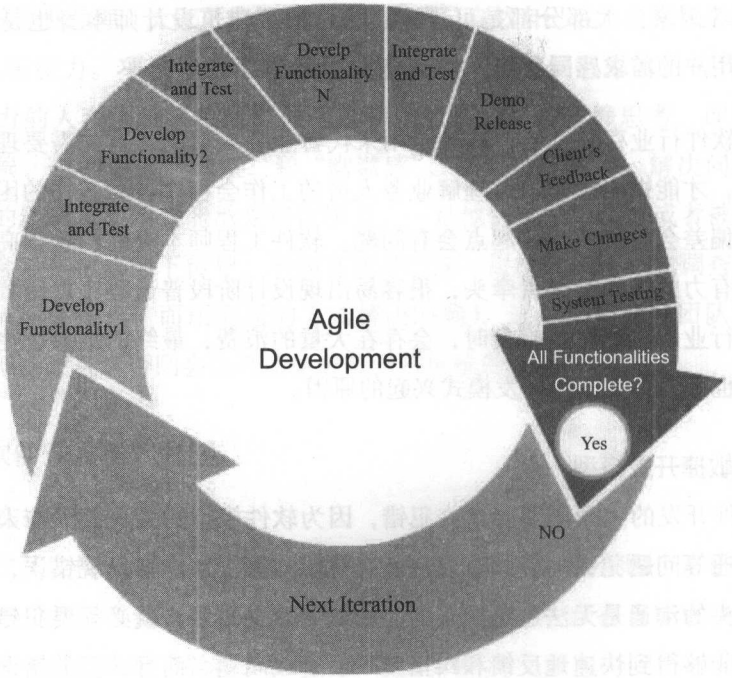
软件开发的生命周期要允许犯错，因为软件模拟的是人，人与人合作总是会因为沟通等问题犯错的。只有通过亲身体验并操作才能够发现错误，仅仅依靠书面和口头的沟通是无法避免错误的。所以要减少错误，就必须要犯错误，只要犯的错误能够得到快速地反馈和纠正就不会造成问题。而且，犯的错误越多，纠正

得越快，就越能够减少线上的问题。要想快速发现和纠正错误，就要做好迭代，而做好迭代就必须确保迭代的生命周期足够短。这就是所谓的“敏捷”，核心就在于快速迭代并迅速反馈。

敏捷软件开发（Agile Software Development）兴起于20世纪90年代中期，最早是为了与传统的瀑布软件开发模式相比较，所以当时称为轻量级方法（Lightweight Method）。二十一世纪初，17位该方法的倡导者建立了敏捷联盟（Agile Alliance），并将该软件开发方法命名为敏捷软件开发过程。

敏捷联盟在成立之初总结了四条基本的价值原则：

- 人员交流重于过程与工具（Individuals and Interactions over Process and Tools）；
- 软件产品重于长篇大论（Working Software over Comprehensive Documentation）；
- 客户协作重于合同谈判（Customer Collaboration Over Contract Negotiation）；
- 随机应变重于循规蹈矩（Responding to Change over Following a Plan）。



整个过程中夹杂了很多在敏捷开发确定之前已经出现的软件开发方法，包括极限编程、Scrum、特征驱动开发、测试驱动开发等。这些方法在敏捷软件开发流程的各个阶段都有充分的体现和应用。

例如，Scrum 主要着重于项目管理，团队中的项目经理需要在每个客户需求到来的时候制定 Sprint 的周期，定义每个 Sprint 的目标、分派任务、进行监督，最后总结得失并开始计划新的 Sprint。相反，特征驱动开发和测试驱动开发主要被应用于 Sprint 周期中，如果项目正处于开发新功能时期，这个阶段主要推行特征驱动开发，所有测试和开发人员都将自己的工作重心放在新的功能上面，从开发和测试两个方面来完成各自的任务。如果项目正处于测试新功能时期，这个阶段需要将工作的重点挪到测试上来，所有的测试和开发人员都密切关注着当前版本的缺陷状况。

3.1.6 常用研发管理体系介绍

不管是敏捷模型还是传统的瀑布开发模型，研发的源头都是需求。需求的好坏，直接决定了开发、测试团队的工作是否存在价值。

前面提过，软件的整个生命周期可以分为软件开发生命周期和软件运行生命周期两种形式。软件开发生命周期的主体是软件开发，软件运行生命周期的主体是软件本身。所以软件运行生命周期才是核心生命周期，因为软件运行生命周期的主体和大的生命周期一致。我们本节主要讨论的是软件开发生命周期，对于这一个阶段，目前有很多种管理方式，我们只简单列举 CMMI 和 RDMS 两种。

1. CMMI

CMMI[⊖]（能力成熟度模型集成）是一种过程改进的方法，可用于指导一个项目、一个单位，或一个企业的过程改进。CMMI 是美国国防部委托卡内基梅隆大学软件工程学院开发出来的，是一个针对产品与服务开发的过程改进成熟度模型。它包含开发与维护活动的最佳实践，涵盖产品从起始到交付与维护的生命周期。

⊖ C 代表 Capability（能力），M 代表 Maturity（成熟度）和 Model（模型），I 代表 Integration（集成）。

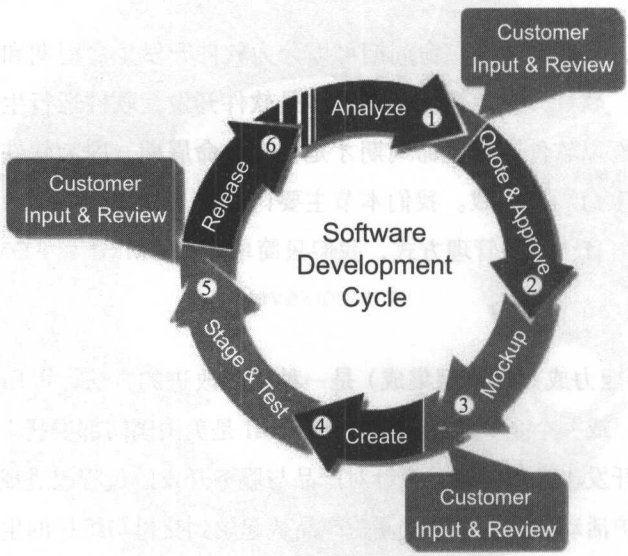
CMMI 包括 3 个模型[⊖]：

- 开发模型，即 CMMI_DEV
- 采购模型，即 CMMI_ACQ
- 服务模型，即 CMMI_Service

CMMI 成熟度等级如下表所示。

级别	说明
初始级	组织通常没有提供稳定的环境维持过程，这些组织的成功，往往依赖组织成员的能力与英雄主义，而不是使用一套经过证实的过程
管理级	可确保组织的项目是按照方针策划与实施过程，项目雇用具备技能的人员，并给予足够的资源，产出可控制的产品，纳入相关的干系人，监督、控制与审查，以及评估遵循过程说明的程度。项目依计划执行和管理
定义级	建立与改进组织标准过程，标准过程是用来确保组织的一致性。项目按照组织定义的过程执行
量化级	组织与项目针对质量与过程绩效建立量化目标，并将它们作为管理过程的准则。以统计的术语了解质量与过程绩效，并在过程生命周期受到管理
优化级	组织根据过程变异共同原因的量化了解，持续改进它的过程

2. RDMS



⊖ 注意，模型不是过程，只说明做什么，不说明如何去做、谁去做。

RDMS（Research and Development Management System，研发管理体系）致力于公司的产品开发并提供最佳的实践及指导方法，适用于公司产品开发全生命周期的管理。研发管理体系框架一般包括概念、定义、设计、实现、验证、移交、维护等多个步骤。

3.2　产品开发过程管理

产品研发生命周期是指创造产品的过程，通常各个阶段按顺序排列且互不交互。完成阶段工作的标志称为里程碑。里程碑包括 4 个要素：时间点、标志性事件、交付物、关闭条件。到达里程碑后要对项目进行评估，确定是按计划继续执行，还是进行必要的变更和调整。

一般情况下，企业开发软件时会按照基线和定制两块并行方式执行项目开发工作。无论什么公司，都需要遵从一套成熟的产品研发过程体系，才能做出质量较好的产品。因此，如果出现项目较多的情况，应该合理地安排基线和定制之前的里程碑，让基线产品能够尽量多地收集用户的通用型需求，为定制项目进度提供技术支撑，减少定制项目中大量更改代码，甚至新增模块等情况发生。此外，产品研发过程体系也需要随着业务实际实现要求变化，不要拘泥于瀑布方式或敏捷方式，凡事都需要找到契合自己的方式。鞋合不合脚，只有脚知道。

为了有效生成某些重要的可交付成果，在需要特别控制的位置将项目分开，就形成了项目阶段。项目生命周期中的各个阶段通常按顺序排列，但有时可能会存在交互的各项目阶段的集合。因为各个项目阶段的工作重点不同、涉及组织不同、人员技能不同，所以根据实际情况将项目划分成合乎逻辑的一些子集，会有助于管理、规划和控制项目。

我们这里以一个基线产品开发过程为例进行介绍。需要注意的是，在项目执行前要明确各个阶段的目标，制定计划、及时沟通，并确保各个阶段所有成员对项目理解一致。

3.2.1 项目启动会

1. 目标

项目启动会的目标是明确该产品开发项目的目标。目标不是孤立存在的，它与计划相辅相成，目标指导计划，计划的有效性影响着目标的达成。所以在执行目标的时候，要考虑清楚自己的行动计划，知道怎么做才能更有效地完成目标，否则，目标不清晰或是过高，都会影响项目的实际结果。

2. 准备工作

在项目启动会之前，我们需要和用户沟通并明确用户需求。注意，本书我们更多侧重于产品基线开发，所以一般不和外部用户直接沟通，这里所说的用户更多是公司内部的产品经理。

项目启动会需要说明项目目标、阶段划分、组织结构、管理流程等关键事项，并将这些内容写入 PPT（最好是有固定格式和范文，让团队内部或者公司内部共同遵守），注意，这些事项需要达成一致意见。对于关键角色任命，事前也需要听取相关领导和项目主要干系人的意见。

参加项目启动会的人员包括部门高层领导、外部主要干系人，还包括团队资深经理、核心人员。

3. 过程及议题

项目启动会一般由开发经理主持，议程如下：

- ❑ 介绍议程和来宾；
- ❑ 介绍项目目标、阶段计划、管理方法；
- ❑ 发布项目的组织结构图；
- ❑ 确认关键角色及其职责，并作出承诺；
- ❑ 参会人员针对介绍内容进行提问交流；
- ❑ 高层领导做项目动员发言，激励和鼓舞士气；
- ❑ 各个相关部门领导表态支持项目工作。

项目启动会后，会议的内容需要整理成《项目启动会纪要》。纪要中记录了项目的启动过程、项目组成员的承诺、各级领导的明确表态等。这份资料作为项目组的第一份正式公告发布，同时宣布项目正式启动。

意见的统一，是项目成功的关键，项目启动会、项目例会有助于统一项目团队的思路，使团队更为团结，也有助于提高团队士气。

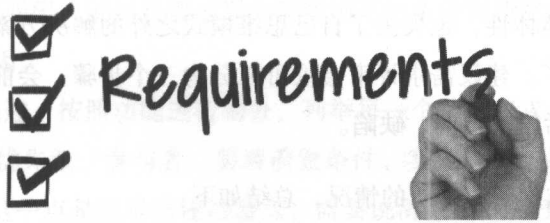
4. 经验和教训

启动阶段多方面工作混杂进行，开发经理容易忙乱，这个时候，需要迅速建立组织架构、明确分工，大家分头工作，然后再着手制定后续详细的项目计划。

项目启动会准备阶段，最好请一个有经验的专家一起参与，帮助开发经理落实很多关键环节，也需要组织团队内部进行讨论，确保在项目启动会上能够流利地回答参与领导提出的各种问题，包括技术类、产品类，这样有利于启动会之后各项工作的顺利开展。

项目启动会非常重要，其意义不仅在于宣布项目启动，而且是确认人员到位的关键时间节点。会议邀请涉及部门的高层领导出席和表态，可以让项目后续的推进获得广泛的支持。项目启动会一旦顺利通过，需要立即着手进行产品需求讨论、编写。

3.2.2 用户需求



软件开始开发前需要确定投入与产出的比值，也就是 ROI（Return On Investment，投资回报率），一旦确定需要创建，就需要安排一系列的资源来支撑这个软件的生存，

这是需求的最原始描述。

为什么既要有用户需求，也要有产品需求？因为两者是有差异的，用户需求由用户提出，对技术一般不描述，只描述产品目标。产品需求是根据用户需求转化而来的技术实现需求，需要针对用户提出的产品目标进行细分，总结出具体的功能点，再针对每一个功能点细分为各种不同的操作流程，对每一个操作流程进行技术化定义。用户需求和产品需求容易产生差异，这是因为虽然大家都在谈需求，但是出发点可能不同，造成了双方关注点和思维方式不同。用户需求关注的是系统如何支持业务流程，背后的需求是“实现业务目标”。技术人员关注的是合理技术方案，背后的需求是“工作量”“实现难度”和“系统性能”。

需求理解不一致

很容易出现的一种情况是，产品经理和技术人员不在一个频道上，所谓的征询意见完全是牵引式问答，你得到的永远是沉默式认同。

此时，整个方案的水平完全取决于讲解者一个人的水平，而对于一个非常复杂的方案，一个人是基本搞不定的，哪怕是非常优秀的产品经理。因为只有讨论才能产生灵感，才能发现自己未曾发现的需求，这就是头脑风暴的优势。

召开产品或者技术方案会议的时候，大家最容易犯的错误就是，自己先绘制了原型图，甚至产品架构图，直接来讲解自己的方案。这是一个完全错误的会议方式，基本上浪费了所有人的时间。离开这个会议室，懂的人懂，不懂的人还是不懂。当不懂的人去执行的时候，就需要摸着石头过河，再局部找你沟通，从而失去了大局观和整体性，也失去了自己思维模式之外的解决方案。于是，二流的产品就这样出现了。建议需求评审会议可以分为三个步骤：会前提供材料，会中讲解、讨论，会后解决疑问、缺陷。

针对这类需求理解不一致的情况，总结如下：

1）开任何会议，一定不要上来就讲解方案，应该先讲解场景。每个场景先讨论背后的需求并定义清楚，这也是软件本身的意义——模拟业务；

2）大家一起讨论，研究这个场景背后的需求是不是最佳场景，能不能删除这个场景，或者是否有其他场景可以代替这个场景同时还能满足背后的需求；

3）如果是需求的最佳场景，再开始讨论这个场景下的产品解决方案；

4）这个时候你打开事先设计好的产品原型图、产品流程图、状态图、用例图，你会发现有很多需要修改，修改之后就是大家一起生成的方案。

3.2.3　产品需求

我们需要弄清楚产品经理或项目需求提出者为什么要做这个项目，这是最本质的业务需求分析。需求分析确定的产品需求，都是从业务需求推导出来的，都必须为业务需求服务。

1. 产品需求编写

产品需求一般包括产品需求规格说明书和产品需求矩阵。

产品需求矩阵一般按照子系统、功能集、执行单元的结构列出所有的功能需求，每列则对应每项功能的工作步骤以及每个步骤的工作量。

产品需求规格说明书一般包括以下几部分：

1）简介：主要由产品功能和定位简介、常用产品和技术术语、本文参考资料等三部分组成。

2）产品描述：包括产品介绍、产品范围，以及产品遵循的标准等。

3）技术约束和局限：说明实现过程中需要满足的条件和所受到的技术方面的限制，并且说明原因。

4）需求详细描述：按照功能进行划分，列举每一个功能需求，说清楚每一个功能的需求描述、优先级、参与者、实现前置条件、实现后的结果、功能执行过程中的正常过程（这一点是需求描述的重点，需要说清楚当没有任何错误发生时，参与者与产品的交互过程。这一过程展示了本功能的核心价值，每一个用例必须要有正常过程）、可选过程（另外一种正常过程）、异常过程（该过程一般会结束整

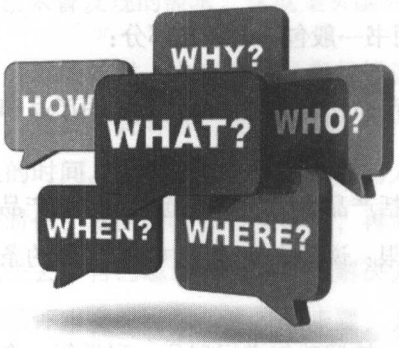
个用例的正常执行）、特殊需求（一些非功能需求描述）、附加说明（与其他需求的相互关系说明）。

5）接口需求：一般包括用户接口（提供用户使用产品时的接口需求）、硬件接口（指出每一个硬件接口的逻辑特点）、软件接口（其他软件调用接口）、通信接口（指定各个通信接口）等。

6）质量需求：一般包括性能需求（产品支持的用户数量、运行速度、对资源的利用率，以及正常情况下和峰值情况下的处理能力等）、可靠性需求（例如平均可用时间、平均故障时间及修复时间、准确性和精确度、错误或缺陷率）、可用性需求（要从用户使用的合理性和方便性等角度进行描述）、可维护性需求（例如行业标准、编码标准、开放性设计架构等所有有利于可维护性或可扩展性的需求）、安全性需求（保护产品的要素，防止各种非法的访问和使用）、可移植性需求（产品从一种环境移植到另一种环境所要求的用户程序、接口兼容方面的约束）、可测试性需求。

7）用户文档需求：包括用户指南、联机帮助手册、安装指南、配置文件等。

2. 需求评审



在需求评审阶段，邀请产品、开发、测试相关人员进行需求评审，产品需求评审主要针对以下几个方面进行：

- WHY：为什么做这个需求？
- WHAT：需求的价值是什么？

- WHEN：需求期望什么时候上线？截止日期？
- HOW：需求是否完整？正常场景是什么？异常场景是什么？技术上分别怎么应对？

例如，产品、技术详细评审需求是否完整，产品功能的正常场景是什么？是否形成闭环？异常场景是什么？是否考虑周全？

需求评审后，由开发和测试负责人分别编写技术方案和测试用例。接下来进行技术方案评审，由开发负责人与其他相关系统的负责人一起讨论，技术方案中必须要有业务流程图和时序图。业务流程图是为了梳理开发对业务的理解，确认是否和需求一致。时序图是为了梳理本次需求涉及的系统交互。技术方案评审通过后，确认工作量和交付时间，反馈给产品经理。

这个过程是需要进行反复讨论的，不管讨论的多激烈，最终能够解决问题才是最重要的。

3.2.4 总体设计

设计阶段的目标主要是对待开发系统的架构进行分析和设计，并建立系统架构的基线，以便为之后的实施工作提供一个稳定的基础。

设计阶段包括了系统架构的输出，一个好的系统架构设计可以帮助人们梳理业务逻辑并抓住核心需求，设计稳定可扩展的业务系统，评估业务开发周期和开发成本，有效地规避风险。例如盖房子的时候得有建筑图纸，有了图纸，才能核算施工周期。

总体设计是整个系统的框架型设计，意义极其重大，一般情况下不能省略（只有维护项目可以省略总体设计，因为基准项目已经设计完毕），所有的产品开发项目均需要首先进行总体设计，它是设计首要步骤，决不允许本末倒置，不能出现先编码后设计的情况，这是软件开发的第二大痛点（第一大是需求不明确、任意变更需求）。

总体设计分为三个阶段：

第一阶段：初始设计。在对给定的数据流图进行复审和精化的基础上，将其转化为初始的模块结构图。

第二阶段：精化设计。依据模块“高内聚低耦合”的原则，精化初始的模块结构图，并设计其中的全局数据结构和每一模块的接口。

第三阶段：设计复审阶段。对前两个阶段得到的高层软件结构进行复审，必要时还可能需要对软件结构做一些精化工作。

1. 总体设计说明书格式

总体设计说明书一般包括以下几部分：

1）产品描述：简述产品的主要功能及应用范围、产品的特点。

2）涉及约束：从《产品需求规格说明书》中提取一些需求约束，具体描述最主要的约束，说明产品是如何适应这些约束的。约束主要包括应当遵循的标准或规范、软件和硬件环境约束、接口和协议约束、界面约束、质量约束（例如正确性、健壮性、性能、易用性、安全性、可扩展性、兼容性、可移植性等），也需要描述自己的想法、思路，以及为什么要采取这样的设计。

3）总体设计：产品设计的技术总体介绍，主要包括软件架构及模块划分、模块描述这两部分。

- 软件架构及模块划分：概述整体设计方案、技术原理和构建思路，并阐述其主要特点。要求突出整个设计所采用的方法、软件的技术体系架构（例如 C/S、B/S 等）以及开发软件所使用的技术、工具。如果产品是由多个模块搭建而成，应说明逻辑结构的构成方式（包含网络拓扑图），以帮助说清楚系统结构。
- 模块描述：对软件架构及模块划分进行描述性的说明，具体说明每个模块的功能和组成，为概要设计打好基础。

4）接口设计：包括外部接口和内部接口。外部接口说明同外界交互的所有接口安排，包括与外部软件或硬件之间的接口、与各个支持软件之间的接口关系。内部接口描述内部各子系统或模块之间的调用准则及关系，以及一些可供其他模块使用的公共模块的接口及调用方式。

5）备选方案及方案对比：首先列出备选方案，如果有多个方案，需要分节列出，如方案一、方案二等。然后对总体设计中的主选方案及每一个备选方案进行对比说明，需给出各方案的优劣势分析，具体说明各个方案的优劣势，以及如果主选方案失败，可以采用哪种备选方案及采用该种备选方案的原因。

6）集成要点：对集成要点进行描述，例如总体设计中所划分主要模块之间的集成顺序，集成时的特殊设备、特殊环境等。

2. 设计方针

1）考虑设计优化问题时应该记住“一个不能工作的‘最佳设计’的价值是值得怀疑的”。

2）应该在设计的早期阶段尽量对软件结构进行精化。可以导出不同的软件结构，然后对它们进行评价和比较，力求得到“最好”的结果。这种优化方法可能是把软件结构设计和过程设计分开的真正优点之一。

3）结构简单通常表示为设计风格优雅，同时保持高效。设计优化应该力求做到在有效模块化的前提下使用最少量的模块，以及在能够满足要求的前提下使用最简单的数据结构。

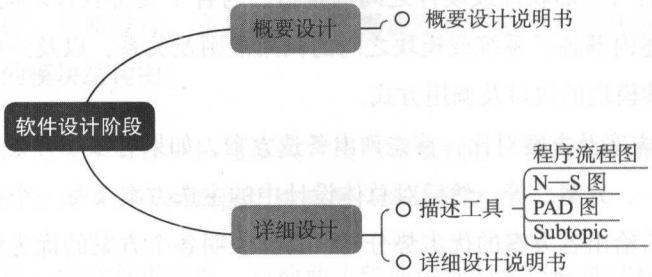
4）模块划分首先要有合理性，这有助于快速对模块认识和理解，可以按照种类（基于逻辑关系、基于功能）来判断划分的好坏（看模块之间的耦合程度和方式，越少越好、越简单越好。有适当的依赖是件好事，证明模块之间有共享和复用，但不建议模块间过度依赖。做到能不耦合在一起就尽量分开来，能不相互依赖就不要相互依赖）。

5）优化时遵守一句格言：“先使它能工作，然后再使它快起来。”

6）从坚实的内核做起：雪球起点不是一堆散雪，而是捏得很紧密的雪核。

7）从小到大慢慢来：一点一点由小变大，而不是通过一次性组装变大。

3.2.5 概要设计



概要设计主要介绍的是设计软件的结构，包括组成模块，模块的层次结构，模块的调用关系，每个模块的功能等。同时，还要设计该项目应用系统的总体数据结构和数据库结构，即应用系统要存储什么数据，这些数据是什么样的结构，它们之间有什么关系。概要设计的目的是描述系统每个模块的内部设计，对总体设计和详细设计承担起承上启下的作用。

概要设计按照结构化设计方法进行设计。结构化设计方法的基本思路是：首先按照问题域，将软件逐级细化，分解为不必再分解的模块，每个模块完成一定的功能，为一个或多个父模块服务（即接受调用），也接受一个或多个子模块的服务（即调用子模块）。模块的概念，和编程语言中的子程序或函数是对应的。

概要设计阶段将软件按照一定的原则分解为模块层次，赋予每个模块一定的任务，并确定模块间的调用关系和接口。

1. 概要设计格式

1）简介：说明编写这份概要设计说明书的目的，指出预期的读者，并对本文会出现的专业术语和术语缩写进行解释。

2）总体架构：一般来说包括系统说明、运行环境说明、基本设计概念、总体结构及模块划分、可测试性设计说明、可维护性设计说明、可配置性设计说明等，也可以增加包含尚未解决的问题列举。这部分需要通过一览表及框图的形式说明本系统的系统元素划分，简单扼要地说明每个系统元素的功能，分层次地给出各元素之间的控制与被控制关系。

3）模块说明：逐一说明各个模块的输入输出及主要处理的功能，可以使用数据流图、状态图、流程图等图形化方法对模块的框架、处理流程进行描述。这里我们需要进一步对每个模块进行划分，并对其子模块进行描述，说清楚其功能及是否是关键模块（测试人员根据该信息确定测试方案）。

4）接口说明：包括用户接口、外部接口和内部接口。用户接口说明将向用户提供的界面、命令、语法结构及软件的交互信息，供用户以及上级模块调用，提供的语法结构应该尽量简单并提供默认值。外部接口说明与外界所有信息传输的安排，包括本系统与各支持软件或外部设备的通信协议及接口函数等，也就是对外部引用接口或协议的调用主要考虑驱动、控制软件以及数据传输等。内部接口说明内部各模块之间的接口，包括类的继承、实现、聚合关系以及各个模块之间如何进行数据交换和共享的方法描述。

5）数据结构设计：包括逻辑结构设计、物理结构设计、数据结构与模块的关系。逻辑结构设计给出所使用的每个数据结构的名称、标识符，它们内部每个数据项的标识、定义、长度以及彼此之间的层次或表格的相关关系。物理结构设计需要给出每个数据结构中每个数据项的存储要求、访问方法、存取单位、存取的物理关系、设计考虑等。数据结构与模块的关系需要说明各个应用程序与访问这些数据结构的各个模块之间的对应关系。

6）系统出错处理设计：说明故障出现后可能需要采取的变通措施，例如热备技术、看门狗技术等。

2. 总结

在这个阶段，设计者会大致考虑并照顾模块的内部实现，但主要仍集中于模块划分、分配任务、定义调用关系。模块间的接口与传参在这个阶段要制订得十分细致明确，需要编写严谨的数据字典，避免后续设计产生误解。概要设计一般不是一次就能做到位，而是需要反复进行结构调整的。典型的调整是合并功能重复的模块，或者进一步分解出可以复用的模块。在概要设计阶段，应最大限度地提取可以重用的模块，建立合理的结构体系，节省后续环节的工作量。

注意，如果是服务端和客户端都存在的系统，概要设计时需要区分服务端概

要设计和客户端概要设计，不能合并在一个文件内。

概要设计文档最重要的部分是分层数据流图、结构图、数据字典以及相应的文字说明等。以概要设计文档为依据，各个模块的详细设计就可以并行展开了。

3.2.6 详细设计

详细设计阶段就是依据概要设计阶段的分解，设计每个模块内的算法、流程，为每个模块完成的功能进行具体的描述，把功能描述转变为精确的、结构化的过程描述。

在详细设计阶段中，各个模块可以分给不同的人并行设计。设计者的工作对象是一个模块，根据概要设计赋予的局部任务和对外接口，设计并表达出模块的算法、流程、状态转换等内容。注意，如果发现需要结构调整（如分解出子模块等），必须返回到概要设计阶段，将调整反映到概要设计文档中，而不能就地解决。详细设计文档最重要的部分是模块的流程图、状态图、局部变量及相应的文字说明等。一个模块对应一篇详细设计文档。

详细设计的目的是描述某一个模块内部的处理流程、开发方法和编码技巧。一般来说，详细设计由项目简介、模块说明（具体说明每一个模块内部的流程、功能、逻辑、消耗以及未解决问题）、接口设计（包括内部接口和外部接口）、数据结构设计（包括物理结构和逻辑结构）、特殊处理等几个部分构成。软件的详细设计，最终是将软件系统的各个部分的具体设计方法、逻辑、功能采用文字方式进行表述。这样在实现过程中，编码人员原则上严格按此进行代码实现即可。详细设计阶段常用的描述方式有：流程图、N-S 图、PAD 图、伪代码等。

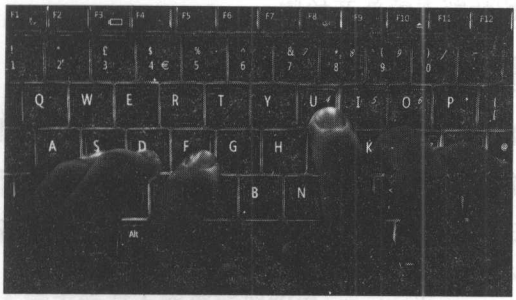
3.2.7 编写代码

1. 编码技巧

（1）优先考虑核心模块的压力测试

在对整个项目做设计、规划的时候，首先要对系统负载最高、开销最大，或

者可能请求量最高的部分编写最基本的数据结构和操作逻辑，然后进行压力测试。也就是说，对一些没有把握且非常重要的部分，提前做压力测试、性能测试，确保都OK，然后正式完成数据结构设计和系统的设计。如果发现有问题，就一直调整，测试到符合预期为止。



我觉得很多程序员，习惯把东西做完，然后等着快上线的时候才做性能测试，如果前面代码实现上出了问题，这个时候就很麻烦了。当然，后期快上线的时候也要做性能测试，但前期的测试我认为还是很重要的。

做好这一点，需要懂一些业务，你要知道业务压力在哪里，业务请求的重心在哪里，很多时候，产品经理不主动告诉你，你也要问清楚。

（2）确保过程可控

我们以日志分析为例：

第一，先从小数据量开始执行测试，验证分析的代码和测试数据返回结果，验证是否符合预期，策略和步骤是否合理。

第二，代码执行时定时跟踪，比如说，每处理10万条日志，写一条状态日志，记录处理的日志条目数和当前的执行时间，有的时候也记录一下资源开销（有时候跑不完资源会崩溃掉，比如内存溢出，这时候需要回访一下，评估一下并想想如何优化调整）。这个执行时间记录也很关键，基本上可以在执行的时候，预估大概的结束时间，中间时间可以比如出去喝个咖啡，或者和别人讨论一个需求。如果一个分析程序跑着，一直没有进度展示，又不知道几个小时可以跑完，

可能会非常浪费时间。

第三，断点可续，因为日志的数据量非常大，内存有大量的中间数据需要保存，有时候，代码跑着跑着，内存就崩溃了。如果每次都从头去跑，那反复花费的时间还是相当长的，所以，要分析程序，必要的话尽可能做到可以在中断的地方，或者上一个可控的断点重新开始跑。

（3）预留地方写注释

很多时候，对自己写的代码也不是很满意，比如某个处理效率不够优化，某个处理的方法不够简洁，或者扩展性比较差，代码写的很弱智，但可能短时间没有办法想清楚最合理的解决方案，考虑到上线初期这里并不需要重点关注，所以也不会特意去优化它，但这种情况下往往需要提前写上注释，并说明下一步优化的可能思路是什么，或者想到的可行方案是什么。

当然，可能超过一半的预留注释都没有真正被改进过，但一旦需要修改，这个注释就很重要。因为时间一长，当时编写的逻辑、思路可能会不记得了，需要调优或者改进的时候，发现以前已经有预留方案，或者能了解当初设计的原委，这些都是很有帮助的事情。

（4）用人人看得懂的逻辑

我们经常说代码的可读性。我的一位朋友毕业到一个公司写程序，他很喜欢写特别绕、特别啰嗦的逻辑，觉得自己很出色、成就感满满，却让接手的人看着一头雾水。后来他慢慢意识到那些啰嗦的逻辑其实都是坑，性能低下、毫无意义，而且非常难以维护，再往后就越写越简单。这好比我们说话，把逻辑复杂的长句，分成几个简单的短句，这样更容易理解，也更容易表达。写代码也是同理，很复杂的逻辑，拆解一下，分成几个简单的逻辑写出来，很清楚，也很有效率。

（5）不要沉迷于框架

很多人写程序，言必称框架，似乎没有框架就写不了程序，其实没必要。比如，就常见的 PHP 来说，本身就可以认为是一种框架了，你再套一层，相当于伤

害了这门语言的活力。

团队开发，用框架也不是说不可以。但你要知道你需要的是什么，你遇到的障碍是什么。

框架最大的问题是什么？是过于繁冗的嵌套。为什么我不建议用框架？因为经常遇到这样的情况，需要对一秒钟几千次请求的处理场景进行调优，此时我只能从数不清的框架中寻找数据处理的逻辑，寻找性能卡点，可能改动代码只有两行，但是找问题需要两天。

框架并不是越高端越好，所谓高端的框架，往往结构更加复杂，而且让人很难理解其中的数据调用过程。

所以，你的技术能力绝对不能被框架约束。

（6）使用熟悉、成熟的技术

听说有这么一家创业公司，因为技术负责人一时兴起，用了MongoDB，然后就进坑了。当然，不是说MongoDB不好，而是相关负责人只是听说还不错，指标很好，然后就兴冲冲地使用了。

现在很多技术人员，习惯性评论技术好坏，你知道这些技术的极限性能吗？如果你用到了极限性能，发现不行，你可以去用别的，或者你的业务场景，确实这些不适合，你也可以去用别的，这都合理。但很多人根本没搞明白自己的障碍和问题在哪里，根本不知道相关技术产品的优势和劣势在哪里，看一堆第三方的数据测评后脑子一热就去学新技术，然后掉进坑里出不来，如果是创业公司，可能项目就死在里面了。所以在使用新技术前，建议全面了解该技术的特征、适用范围，以及不适用的范围。

（7）细节没优化前，别谈架构

系统访问负载升高，系统稳定性较差，遇到这种情况，别急着换服务器或者技术，先把问题搞清楚，看看具体原因到底在哪里，单服务器的瓶颈和压力在哪里，再来谈这个问题。

（8）多留日志

经常会产品上线后遇到诡异的问题，而且核查起来很复杂，回到公司又无法复现。这类 Bug 是最难处理的。所以，建议在代码里面、Web 服务里面，以及系统运维的脚本里面，多增加一些日常的日志输出，对异常进行自动的信息采集。这样，出问题的时候，可以查看问题的信息，对定位问题、分析问题、解决问题帮助都很大。

2. 日志规范

日志作为重现系统场景的手段，对于定位系统问题具有非常重要的作用。我们应该明确在何处打印日志，打印多少日志，而且需要按照严格的格式和统一的风格，绝不能随意。

日志打印少了，开发者会觉得一头雾水，缺少关键信息，就没有充足的证据明确问题。日志打印太多，会影响系统性能，毕竟日志文件也会涉及磁盘 IO 操作，而且日志过多会导致循环覆盖，丢失关键信息。

一般来说，日志组件会采用 Log4j 这样的稳定组件，通过使用这类组件，可以实现以下问题：

1）日志级别动态调整，无须重新编译程序。

2）日志大小可定制，实现循环覆盖，无须定制时可以压缩打包。

日志约束条件一般包括如下：

1）单个日志文件大小限制，可以设置上限、下限，这样有利于日志总大小计算。

2）日志数量限制。

3）日志统一路径、命名规则。

4）日志备份机制设计，可以预防丢失日志。

日志格式一般为：时间 | 函数地址 | 等级 | 模块 | 行数 | 日志详情

日志等级一般分为 ERROR、WARN、INFO、DEBUG 和 TRACE，不同场景下使用不同的日志等级，不能随意使用。

1）严重的系统错误，会导致流程中断或异常流程，需要使用等级最高的 ERROR 等级。例如数据库操作失败、服务异常等场景。

2）异常但不影响系统的正常流程，使用 WARN 等级。例如，查询不存在的数据。

3）系统或流程的关键状态变化，使用 INFO 等级。例如，系统启动或退出步骤执行。

4）业务流程点或调试信息，使用 TRACE 等级。例如，函数执行过程中的调试信息。

5）实际生产场景下日志等级一般是 INFO。

日志详细规范包括：

1）日志语言必须是英文，保证日志是 ASCII 编码。

2）日志中不能打印安全敏感信息，如用户名、密码、序列号等。

3）日志不能暴露系统业务处理逻辑。

4）日志必要时需要记录上下文信息，确保不缺失定位问题需要的关键信息。

5）不同模块的日志格式和风格保持一致。

6）抛出异常后需要把异常详细信息记录到日志中。

7）日志一定要通顺简洁，不能暴露关键函数和变量。

日志数量较多的情况下，可以通过专门的日志分析工具，例如 OtrosLogViewer 进行批量分析。

3. 针对代码编写的十条建议

原则 1——简化控制流程

使用尽可能精简的控制流程构造编写程序——不要使用 setjmp 或 longjmp 构造、goto 语句，以及直接或间接的 recursion。

原因：简化控制流程有助于提高代码清晰度，增强代码可验证能力。不使用递归，便不会产生循环的函数调用图，这样也可证明所有本应有界的执行实际上都是有界的。

原则 2——为循环设置上限次数

所有循环必须有固定次数的上限。我们可以通过验证工具静态地证明，为循环中迭代数量所设立的上限次数未被超越。如果无法以静态方式加以证明，则可认为未遵守该原则。

原因：为循环设置次数界限，避免使用递归，这些做法有助于预防代码失控。然而该原则无法适用于本就不应终止的迭代（例如进程调度器）。此时将沿用该原则的逆向原则：必须能够静态地证明迭代不能终止。

原则 3——不使用动态内存分配

不要在初始化完成后进行动态内存分配。

原因：诸如 malloc 等内存分配机制，以及垃圾回收器、通常会产生无法预知的行为，进而可能会对性能产生影响。更重要的是，还有可能因为程序员的失误造成内存错误，例如：

1）试图分配超过可用物理内存数的内存。

2）忘记释放内存。

3）继续使用已被释放的内存。

4）对已分配内存进行越界使用：应强制所有模块位于固定大小、预先分配的存储区域中，借此可避免此类问题，并简化内存使用情况的验证工作。

对于堆中未分配内存的情况，动态请求内存的唯一方式是使用栈内存。

原则 4——不使用冗长的函数

任何函数的长度不应超过使用标准参考格式（每个声明最多一行，每个语句最多一行）打印的纸张上一页纸所能容纳的字符数。这意味着函数的代码不应超过 60 行。

原因：过长的函数通常意味着结构并非最优。每个函数都应是可理解且可验证的单一逻辑单位。如果在计算机显示器上需要多屏界面才能完整显示，这样的逻辑单位通常会极难理解。

原则 5——低断言密度

程序的断言密度（Assertion Density）应平均保持为每个函数最少两个断言。断言可用于检查现实运行过程中本来绝不应出现的异常状况，因此应定义为 Boolean 测试。当断言失败后，应执行明确的恢复操作。如果静态检查工具证明断言绝对不会 Fail 或 Hold，则可认为未遵守该原则。

原因：业界的代码编写工作统计报告显示，通过单元测试可发现，通常我们所编写的每 10 ～ 100 行代码中至少会存在一处缺陷。随着断言密度的增高，拦截缺陷的机会也会增大。断言的另一个重要之处在于，它是防御性编程（Defensive Coding）策略的重要组成部分。我们可以使用断言验证函数执行前后的状况，函数的执行参数和返回值，以及循环不变式（Loop-invariant）。在完成性能关键代码的测试工作后，可将断言选择性地禁用。

原则 6——以最小范围级别声明数据对象

该原则同时也是数据隐蔽（Data Hiding）的基本原则。所有数据对象均必须以尽可能最小的范围级别进行声明。

原因：如果某对象不在范围内，意味着其值将无法引用或已损坏。该原则不鼓励使用多种可能导致故障诊断工作变得更复杂的互斥意图重用变量。

原则 7——检查参数和返回值

应在每次调用函数后检查非空函数的返回值，并应在每个函数内部检查参数的合法性。在最严格的形式下，该原则意味着就算 printf 语句和文件 close 语句的返回值也应进行检查。

原因：如果对一个错误结果的响应与对成功结果的响应本没有任何区别，那么很明显需要检查返回值。通常对 close 和 printf 的调用便符合这种情况。此时一

种可行的方法是将函数的返回值明确抛给 void，这意味着开发者明确（而非意外地）决定忽略该返回值。

原则 8——限制预处理程序的使用

预处理程序（Preprocessor）应仅限用于头文件和宏定义。递归的宏调用、令牌传递，以及变量参数列表均不允许使用。就算在大型应用程序开发工作中，标准样板文件（Boilerplate）之外也可能有必要使用一两个以上的条件编译指令，这是为了避免将同一个头文件包含多次。这种用法必须通过工具检查器添加标记，并通过代码阐述原因。

原因：C 语言预处理程序是一个强大但较为含糊的工具，有可能会彻底破坏代码的清晰度，并让很多基于文本的检查器产生混淆。就算具备正式的语言定义，包含无界限预处理程序代码的构造也会显得非常难以解读。有关条件编译的注意事项同样很重要。就算只使用 10 个条件编译指令，代码也可能会产生 1024（2^10）个版本，这会导致测试工作量剧增。

原则 9——限制指针的使用

指针的使用必须加以限制。通常只允许不超过一层的解引用（Dereferencing）。指针解引用操作不应隐藏在 typedef 声明或宏定义内部。此外函数指针也是不允许使用的。

原因：指针很容易被滥用，就算专家也难以彻底避免。指针的存在会使得我们难以跟踪或分析程序中数据的流动，尤其是在使用基于工具的静态分析器执行这些操作时。函数指针还会对静态分析器所能执行的检查类型产生限制，因此除非有非常必要的理由，否则一般情况下不推荐使用。如果使用函数指针，通常无法通过工具证明递归的缺席，此时只能提供其他方法弥补这种分析能力的缺失。

原则 10——编译所有代码

从开发工作第一天开始时，就必须对所有代码进行编译。必须启用编译器的警告功能，并使用最细致的检查选项。代码必须能通过这样的设置，在不产生任何警报的情况下顺利编译完成。所有代码必须每天一次，使用至少一种（多种则

更好）最新型的静态源代码分析器进行检查，并且必须顺利通过分析器的整个检查过程而不产生任何警告。

原因：市面上有很多效果卓越的源代码分析器，其中很多甚至是以免费软件的形式发布的。对于这样可以直接使用的现成技术，任何软件开发工作都没理由不加以充分利用。

3.2.8 代码审核

众所周知，在团队中进行代码审查（Code Review）可以提升代码质量，分享项目知识、明确责任，最终构建更好的软件、更好的团队。当然也有许多方法可以进行代码审查，例如在 GitHub 中提到的 Pull Request，或使用像 JetBrains 的 Upsource 之类的工具。然而即使拥有清晰的流程和正确的工具，还是遗留了一个大问题需要解决——我们需要找寻哪些问题。

1. 代码审核重要性

代码审核及其重要，一般来说每周都要做一次代码审核。代码审核有利于跟踪项目进展情况，我们能真实地看到其他人的工作进展如何，并且能更早发现他们是否误入歧途。有时候，手下人会说“完成得差不多了！”你去看代码时发现什么都没有，诸如此类，总之离完成还很遥远。在管理中，这种情况是最让人讨厌的，所以我认为代码审查是避免这种麻烦的最佳途径。

2. 代码审核内容

一般来说，代码审核内容由如下部分组成：

1）前置条件：代码是否可以正常运行、开发工具或者运行过程当中有没有严重警告灯。

2）代码规范性：主要由注释、排版、命名规则等组成，注释又可以进一步分为类定义、函数头、函数内代码实现等相关注释说明，以及注释风格统一要求。排版主要包括文件顺序（各类变量定义、函数实现等）、代码行格式要求（代码行

最多字符数量、运算符外侧空格、缩进等）、对齐（函数、变量、注释等）。命名规则主要包括对于常量、类名、变量名、函数，以及通用命名规则的遵循。

3）代码逻辑：包括函数、控制结构、内存管理、类 / 结构体、其他高级特性、并发处理、设计模式使用、跨平台设计等。

此外，在做代码审核的时候，我们关注的应该是新增或修改的代码行数，而不是整个库的代码量变化。

3. 代码审核关注点

1）设计

- ❑ 如何让新代码与全局的架构保持一致？
- ❑ 代码是否遵循 SOLID 原则，是否遵循团队使用的设计规范，如领域驱动开发等？
- ❑ 新代码使用了什么设计模式？这样使用是否合适？
- ❑ 基础代码是否使用了一些标准或设计样式，新的代码是否遵循当前的规范？代码是否正确迁移，或参照了因不规范而淘汰的旧代码？
- ❑ 代码的位置是否正确？比如涉及订单的新代码是否在订单服务相关的位置？
- ❑ 新代码是否重用了现存的代码？新代码是否可以被现有代码重用？新代码是否有重复代码？如果是的话，是否应该重构成一个可被重用的模式，还是当前还可以接受？
- ❑ 新代码是否被过度设计了？是否引入现在还不需要的重用设计？团队如何平衡可重用和 YAGNI（You Ain't Gonna Need It）这两种观点？

2）可读性和可维护性

- ❑ 字段、变量、参数、方法、类的命名是否真实反映它们所代表的事物。
- ❑ 是否可以通过读代码理解它做了什么？
- ❑ 是否理解测试用例测了什么？
- ❑ 测试是否很好地覆盖了用例的各种情况？它们是否覆盖了正常和异常用

例？是否有忽略的情况？

- 错误信息是否可被理解？
- 不清晰的代码是否被文档、注释或容易理解的测试用例所覆盖？

3）功能

- 代码是否真的达到了预期的目标？是否有自动化测试来确保代码的正确性，测试的代码是否真的可以验证代码达到了协定的需求？
- 代码看上去是否包含了不明显的 Bug，比如使用错误的变量进行检查，或误把 and 写成 or ？

4）其他

- 是否需要满足相关监管需求？
- 是否需要创建公共文档或修改现存的帮助文档？
- 是否检查了面向用户的信息的正确性？
- 是否有会在生产环境中导致应用停止运行的明显错误？代码是否会错误地指向测试数据库，是否存在应在真实服务中移除的硬编码的 stub 代码？
- 对性能的需求是什么，是否考虑了安全问题？

3.2.9 单元测试

1. 什么是单元测试？

要认识单元测试，首先要明白什么是“单元（Unit）”。所谓“单元”指的是代码调用的最小单位，实际上指的是一个功能块（Function）或者方法（Method）。所以单元测试指的就是对这些代码调用单元的测试。

单元测试是一种白盒测试，就是必须要对单元的代码细节很清楚才能做的测试。所以，单元测试的编写和执行都是由软件工程师来做的。除了单元测试，还有集成测试。集成测试基本都是黑盒测试，主要是由测试人员根据软件的功能手册来进行测试，需要有专门的测试环境配合。集成测试又分功能测试、回归测试等。

2. 单元测试质疑

有人会问，如果方法就是单元，一个软件包含那么多方法，那单元测试岂不是要写很多？确实会存在这个问题，这也是为什么很多人认为单元测试不可行的原因之一。

另一个原因是，一提单元测试就有人提出必须要做模拟（Mock）。因为要把代码跑起来就必须要模拟上下文，这样代码才可以跑起来测试。

这么多单元要测，并且跑起来还要那么多外部环境要模拟，仅维护这么多测试就要比开发量还要大了，这不是找罪受吗？更不要说单元测试也是代码，单元测试本身是不是也要测啊？因此很多软件工程师反感单元测试，即便有强制规定也是敷衍了事。

3. 如何做单元测试

需要单元测试的代码实际上是开发人员自己写的逻辑，测试逻辑所依赖的环境是否正常不是单元测试的目的。在环境访问代码中引入逻辑，只会让逻辑更难测试，导致逻辑代码无法进行单元测试。因此，可单元测试的代码，才能够采用单元测试。有一个方法可以判断是否为可测试的代码，就是看这个方法能否用一个 main 函数直接运行，如果可以的话就是可单元测试的代码。可测试的代码还有另一个特征，就是该方法单元的参数，开发人员可以自由模拟，不需要依赖外部环境。

如果代码里有逻辑，但是不可单元测试的话，就需要改造代码。改造的方法，就是要确保逻辑代码和外部环境相关代码隔离，这个逻辑代码就是可单元测试的。而隔离的办法就是把代码的执行顺序，也就是单元的执行生命周期，做架构的拆分。访问代码的核心就在于传递上下文的数据，因此先把逻辑需要的数据从上下文环境中取出来，给逻辑代码单独另建一个单元，再把从环境中取出来的参数作为入参传入新建的逻辑代码单元。这样新建的逻辑代码单元就只依赖入参而不再依赖环境，从而使开发人员可以自由地模拟输入参数做单元测试了。

4. 测试用例编写

单元测试主要是由开发人员对自己编写的代码进行自测或相互进行交叉测试，用以检查代码是否符合编码规范，是否存在逻辑错误。

针对程序的规范和逻辑，可以引入自动检查编码规范和编码逻辑代码的检查工具进行测试。

一般来说，单元测试用例主体包括每个模块的详细测试用例、修改记录这两项。修改记录比较容易理解，就是历次变更的时间和内容，而每个模块的详细测试用例的范例如下：

基本信息							
项目名称							
测试内容							
测试人员							
测试时间							
模块	用例编号	测试内容	前置条件	测试步骤（需要列出详细步骤）	预期结果	测试结果（通过/失败/挂起/不需要）	备注

然后自己根据实际用例一条一条地填写即可。

5. 单元测试优点

刚开始编写单元测试代码确实存在一定的代码量，会觉得有点占用时间。但是随着代码的深入，测试工作量会越来越小，因为单元测试做到位了，代码可以直接运行。反观手工测试，刚开始确实很快（需要测试的内容少），随着代码增多，手工测试工作量加大，反而花得时间更多，因为以前测试过的内容还需要重新再测，无法把自己的重复工作自动化。

一旦养成单元测试的习惯你就离不开单元测试了，因为一旦上瘾，会觉得没写单元测试代码就很不安。大部分时候，单元测试覆盖了100%，只要业务没有太大的变化，单元测试就不需要维护，即一次投入会持续受益。

3.2.10 集成测试

1. 什么是集成测试

集成测试，也叫组装测试或联合测试。在单元测试的基础上，将所有模块按照设计要求组装成为子系统或系统，进行集成测试。实践表明，一些模块虽然能够单独工作，但并不能保证连接起来也能正常地工作。一些局部反映不出来的问题，在全局上很可能会全面暴露出来。

集成测试是在软件系统集成过程中所进行的测试，其主要目的是检查软件单位之间的接口是否正确。即根据集成测试计划，一边将模块或其他模块组合成越来越大的系统，一边运行该系统，以分析所组成的系统是否正确，各个组成部分是否合拍。集成测试的策略主要有自顶向下和自底向上两种。也可以理解为在软件设计单元、功能模块组装、集成为系统时，对应用系统的各个部件（软件单元、功能模块接口、链接等）进行的联合测试，以决定他们能否在一起共同工作，部件可以是代码块、独立的应用、网络上的客户端或服务器端程序。

2. 集成测试困局

目前很多公司将测试环节推到了集成测试环节，交由测试人员来完成集成测试。这就形成了一个架构拆分，形成了新测试的生命周期，由测试人员来推进测试的生命周期。

大部分软件开发团队仍然完全依赖于人工的集成测试来提升软件质量。而集成测试属于新的分工，需要大量的人工介入，需要比较长的时间周期才能完成测试，也增加了很大的沟通成本。如果软件不经常进行迭代，人工测试问题还不太大，毕竟难得的一次。但是现代的软件开发迭代速度非常快，经常是按周的频率进行迭代。如果每周都要全面的测试，那就要维护一个比开发团队大得多的测试团队。测试因此就成为了迭代速度的瓶颈，会导致迭代速度的减慢，损害软件的更新以及业务的增长。

为了提升集成测试的效率，集成测试也可以进行大量的自动化。比如回归测

试就是自动化的重点。另一个提升测试效率的办法，就是让测试回归到软件开发工程师职责范围内，也就是单元测试。当然，单元测试也属于自动化测试。

3. 集成测试重点

集成测试重点是模块之间的连接。

将经过单元测试的模块组装成完整的程序。工作任务包括制定集成测试策略，确定集成测试步骤，设计集成测试用例，然后逐一添加模块进行测试。集成测试由测试人员负责，应该在概要设计完成后进行设计工作，并在单元测试完成后执行。

所有的软件项目都不能摆脱系统集成这个阶段。不管采用什么开发模式，具体的开发工作总得从一个一个的软件单元做起，软件单元只有经过集成才能形成一个有机的整体。具体的集成过程可能是显性的也可能是隐性的。只要有集成，总是会出现一些常见问题，工程实践中几乎不存在软件单元组装过程中不出任何问题的情况。一般来说，集成测试需要花费的时间远远超过单元测试，直接从单元测试过渡到系统测试是极不妥当的做法。

4. 集成测试优点

- 单元测试具有不彻底性，不能保证模块间接口信息内容的正确性，以及各模块间相互调用关系是否符合设计，只能依靠集成测试来保障。
- 同系统测试相比，由于集成测试用例是从程序结构出发，目的性、针对性更强，测试项发现问题的效率更高，定位问题的效率也较高。
- 能够较容易的测试到系统测试用例难以模拟的特殊异常流程，从纯理论的角度来讲，集成测试能够模拟所有实际情况。
- 定位问题较快，由于集成测试具有可重复性强，对测试人员透明的特点，发现问题后，很容易定位，所以能够有效地加快进度，减少隐患。

3.2.11 系统测试

系统测试阶段包括系统测试方案和用例编写、功能性测试、性能测试、稳定

性测试。

1. 测试方案和用例编写

测试方案和用例是对业务逻辑的一次梳理，从测试角度完整理解一次需求，看看是否有测试场景遗漏，验证业务场景的完整性。测试方案的侧重点是整个业务方案层面的测试，而测试用例则是根据每一个需求所实现的对应具体测试方式，理论上来说它和每一个需求是N：1的关系，即一个需求对应多个测试用例，为什么会是多个？因为每个需求的实现内部都会有很多参数校验、业务分支、异常情况，需要多个用例才能覆盖完整。测试方案和用例是需要评审的，而且必须经过评审。评审过程需要测试负责人和产品、开发等相关人员一起评审。测试用例评审通过后，用例会分成两块，即核心用例和一般用例。

2. 测试阶段的分工

测试阶段，开发人员的主要工作是修复Bug或者完成突发需求，测试人员的主要工作是执行测试用例，发现Bug、提交Bug以及验证Bug。很多场景下，开发人员每修复一个Bug，就会通知测试人员进行验证，这种工作方式的效率非常低下。在高效的研发体系中，回归测试是有时间约定的。例如：约定早上9点半提交测试，测试人员进行第一轮测试，然后集中提交Bug到Bug管理系统，开发人员收到指派Bug后，开始进行修复。下午3点，约定第二轮回归测试，由测试人员对已经修复的Bug集中进行验证，有问题，则再次提交Bug，指派给开发人员修改。下午7点，约定第三轮回归测试。在这样的体系下，开发人员专注于修复Bug，到了约定的时间，测试人员集中测试，减少了不断提测、部署环境浪费的时间。

3. 功能性测试

功能性测试环节的目标是为了验证需求分析确定的功能是否齐全并被正确实现，同时还要对安装、部署、适应性、安全性、界面等非功能性需求进行测试。

功能性测试一般由独立测试小组采用黑盒方式来测试，主要测试系统是否符

合《产品需求规格说明书》。功能性测试阶段又可分为三个步骤：

1）模块测试：测试每个模块的程序是否有错误；

2）组装测试：测试模块之间的接口是否正确；

3）确认测试：测试整个软件系统是否满足用户功能和性能的要求。

该阶段结束后需要交付测试报告，说明测试数据的选择，测试用例以及测试结果是否符合预期结果。测试人员发现问题之后要经过调试找出错误原因和位置，然后进行改正。功能性测试是基于系统整体需求说明书的黑盒类测试，应该覆盖系统所有联合的部件。以阿里双11的全链路压测为例，所有技术人员达成了一个共识，一定要有一套系统能够最真实地模拟双11当天的流量，能够及时发现大压力下线上系统的所有问题和风险，保障真实场景下的用户体验。

4. 性能测试目标

阿里双11当天交易峰值较平时增长400倍，平日运转良好的系统面对突发的业务流量，所有的问题都会被重新定义。全链路一体化方案通过逼真化模拟实际大促时的流量特点，以自动化的方式评估、优化和保护整个交易链路，确保了双11的稳定性。这就是性能测试的重要性体现。

性能测试验证系统的稳定性和效率，检查系统是否满足规定的性能要求。性能测试通常选择一些典型的功能，检验这些功能在大量用户同时使用系统时系统是否稳定。性能测试由测试人员负责，可以在系统测试完成后进行，也可以对重要模块先进行性能测试，可以贯穿整个测试周期，目的是尽早发现系统的性能瓶颈并提早解决。

性能测试是一个十分复杂的系统工程，对测试人员的能力水平提出了更高的要求，需要性能测试人员具备非常全面的知识与技能，能够定位应用的性能瓶颈，并提出适当的优化方案。

5. 性能测试测试模型与测试指标

在进行场景设计之前我们应该先确定本次性能测试的测试指标与测试模型。

测试指标和测试模型是进行场景设计的前提和基础，是场景的输入。根据被测系统的类型不同，可能测试指标的类型略有不同。对于在线 Web 类的应用，测试指标一般包括在线用户数、最优并发用户数、最大并发用户数、交易平均响应时间、目标 TPS 等。对于接口调用类的应用测试指标一般包括目标 TPS、平均响应时间等。

测试模型就是被测试系统的各交易在线运行时承受的交易数量（或请求数量）的比例而不是并发用户的比例。为什么不是并发用户的比例呢？因为实际用户的操作具有不确定性，使用测试工具很难模拟真实用户的行为。另外，在进行运营数据分析时很难获取用户的操作行为，而应用的交易记录却很容易通过查询的方式获取。应用实际承受的压力是用户的实际操作请求，在线用户如果没有进行实际操作那么他最多将消耗一个连接线程，而应用 CPU 并不会有什么资源消耗。100 个用户平均每个花费 10 秒下一个订单和 10 个用户每 1 秒钟下一个订单对应用带来的压力是一样的。所以，在场景中能用最少的并发用户来模拟真实的请求是最经济的选择方式。

那么，测试模型到底该如何确定呢？通过需求调研获得。下面介绍的两点最常用的调研方式：

- 对于还未上线运营的新系统，一般会让应用的产品经理或负责人给出一个预估的比例，但是这个预估需要我们进行评估，不是随意的。对于一个以提供下单交易为主的应用，通常下单交易是占整个模型的较大比例，如果需求方提出的模型是查询比例较高，那么就有理由怀疑该模型的合理性。对于这种情况，建议选择几个常见的典型的模型来配合需求模型进行场景设计。
- 对于已上线运营的应用，一般会分析实际的交易数据来确定交易比例，这样会更加精准。例如一个应用对用户提供下单、查询、退款三个交易，通过 DBA 在线查询某日的交易数据总量为 200 000 笔，其中交易下单 160 000 笔、查询 38 000 笔，退款 2000 笔，由此预测各交易的比例是 80%、19%、1%，那么这个比例就是测试模型。

6. 稳定性测试

稳定性测试和性能测试都必须等到系统基本没问题、趋于稳定时再进行才有效果，否则很难顺利测下去，出现异常也不能定位究竟是系统架构的问题，还是功能的问题。

稳定性测试（亦可称可靠性测试）通过给系统加载一定的业务压力，让系统持续运行一段时间（一般为 7×24 小时），检测系统是否能够稳定运行。

以一个 BI（商业智能）的例子来进行我们的稳定性测试建模，主要步骤如下所示：

1）软件主要业务：从大量元数据中提取（ETL）客户关心的数据并最终生成报表（本文以微软平台 BI 为例：SSIS，SSAS，SSRS）

2）用户场景：利用 SSIS 包进行 ETL 操作将元数据计算转化后导入到数据立方体（Cube）中。

3）典型负载：每小时 3000 个用户，100000 条数据，执行 7×24 小时。

4）测试环境：需求文档中规定的配置。

5）主要性能指标：

- ETL 时间：9 分钟，差别：1 分钟，方差：<0.1
- 系统相关：CPU，Memory，Private Mbytes/sec 等

6）稳定性指标模型：

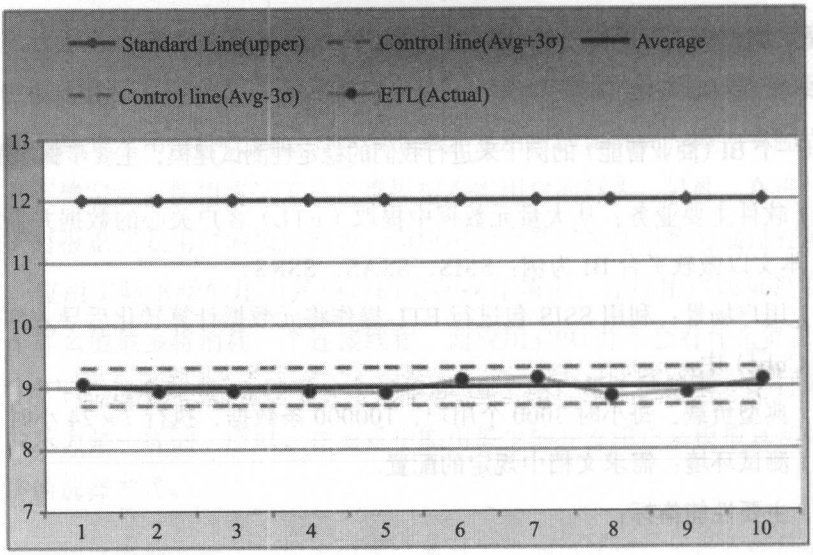
计算公式

$$\text{Variance}:\left(\frac{\sum_{i=1}^{N}\left(X_i-\bar{X}\right)^2}{N}\right)$$

$$\text{Percentage}:\%=\left(1-\frac{\sqrt{\left(\frac{\sum_{i=1}^{N}\left(X_i-\bar{X}\right)^2}{N}\right)}}{\bar{x}}\right)*100\%$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}\left(X_i - \bar{X}\right)^2}{N}}$$

稳定性模型



从图表中可以看出：

- ETL 上限为 12 分钟（即如果超过 12 分钟就证明有瓶颈，需要核查）。
- ETL 平均值为 9 分钟。
- 控制线的上下方分别为 Avg 加减 3 倍的方差。
- 实际使用时间围绕平均值上下分布（标准为同一向不能出现连续 7 个点：如连续 7 个实际检测值都在平均值的上方，这时就需要进行调查）。

稳定性测试是用来验证产品在一定的负载下能否长时间地稳定运行，其主要目的是验证系统能力，在能力的验证过程中找到系统不稳定的因素并进行分析解决。

3.2.12 产品发布

产品发布是系统测试结束后的最后一步，通常在软件产品开发过程中不需要

产品试制环节，可以直接上线，只需要系统测试员输出系统测试报告并批准产品发布（上线）就可以了。

产品发布前需要通过产品发布说明会的形式，对整个产品开发过程从立项开始回溯过程，总结整个过程中的经验教训。这一会议可以通过正式的会议形式召开，需要召集产品经理、主要开发人员、测试人员、上级领导等参与，必须准备充分，尽最大可能说清楚这个产品发布之后的效果、效益，为上线后的价值评估做准备。这一环节不可缺少，即便在互联网公司，迭代速度很快的情况下，这一环节也需要满足。

3.2.13 开发过程复盘

1. 复盘的意义

其实开发过程体系里并没有这一过程，但是我个人认为它非常重要。

所有的总结，只有带着问题去思考才会有收获，这就是复盘。不论我说多少，如果没有类似的经验，就很难有很强的共鸣。我觉得看清一个问题最好的方式，就是你曾经处在一个问题的两个不同的角色中。

当你去问一个老人很多问题的时候，虽然可能很快得到解答，但当有一天，有很多新人来问你问题的时候，你就能体会到什么问题该问，什么问题不该问。

当你自己写代码的时候，别人给你提意见，你可能觉得很好，也可能觉得很烦。等有一天你去读一段晦涩难懂的代码的时候，你才能理解为什么有些细节会那么重要。其实这就是一个成长的过程。努力、用心、思考，这些都需要时间的积累。但如果不用心，时间花得再多，也不会有太大的进步。

总结项目经验教训的目的在于总结问题、分析原因，避免以后犯同样的错误，而不是追究谁的责任。

假设一个需求理解的缺陷，如果在需求阶段发现，修改一下可能只要一个小时，但是如果到了设计完成时发现这个缺陷，因为涉及的人员、文档增多，估计

要一天时间，而如果等到代码都编写完成时才发现这个缺陷，可能需要十天八天了。如果缺陷没被发现，而是直接到了生产系统中呢？这就不是工作量的问题了，估计损失就难以估计了。在质量管理的理论中，缺陷越晚发现，修复的代价就要乘上十倍。

2. 如何复盘

项目经理需要在产品开发过程中召开一个或者多个复盘会议，会议要对项目启动会的每一个环节进行回溯。例如项目启动会是否准备充分，启动会现场是否有提问没有回答上来，为什么没有回答准确；产品需求是否按期完成、评审是否通过、为什么存在产品经理不满意产品需求的情况；总体设计是否提前与产品需求完成；总体设计、概要设计评审是否存在高优先级缺陷，为什么会有这样的缺陷存在，是不是设计阶段没有考虑周全；单元测试是否完成，为什么单元测试效果不好；系统测试阶段发现了哪些高优先级缺陷，冒烟测试是否通过，为什么没有通过。

可以复盘的内容很多，一位严格、高水平的技术经理，这一环节应该是他花费大量时间的环节。复盘的参与人员不用很多，只需要该产品的开发人员参与就可以了。再重申一句，复盘会不要太严肃，它不是“批斗会”，而是为了总结经验，不断优化，不再犯同样的错误。

3.3 产品开发过程杂谈

3.3.1 理解业务的重要性

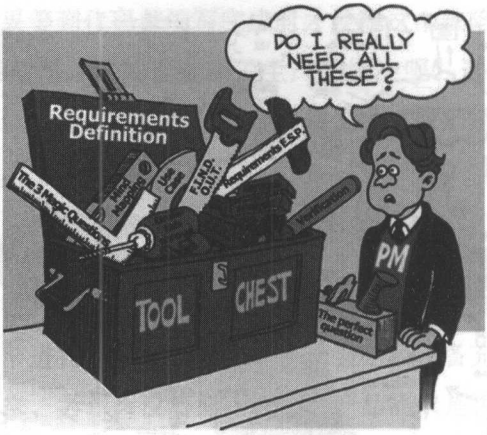
业务问题的本质是业务所服务对象的利益问题。根据业务对象的利益，可以整理出业务的生命周期和生命周期的拆分，再根据业务的概念和组织方式，可以分析出业务的核心生命周期。根据当前业务增长的方式，也可以反过来理解业务的生命周期拆分。通过对生命周期的分析，可以快速理解业务，并进行领域建模，为软件模拟业务做好准备。

为了能够让业务在软件中很好地运行起来，软件工程师必须理解业务的生命周期、拆分方式，以及业务服务对象的利益所在，即业务问题。业务的核心生命周期是什么？这些问题是如何拆分解决的？涉及了哪些概念？这些概念分别解决了哪些问题？针对这些问题，软件工程师经常需要根据自己的理解，创建一套概念体系来表述业务，或者用软件行业的术语来表述业务。

软件工程师还必须要考虑，用什么样的硬件把软件跑起来？怎样才能跑得好、跑得快，同时软件还能随着业务的流量逐渐地长大？

3.3.2 应对需求变更

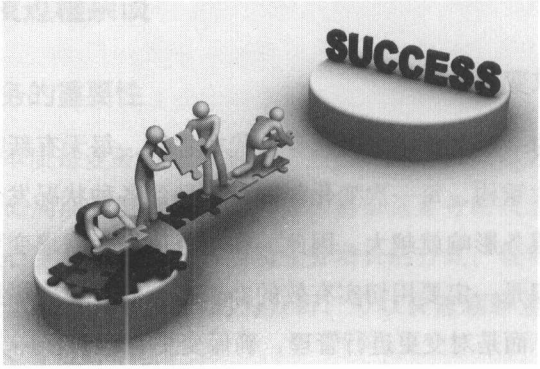
互联网业务的一个主要特点是业务迭代非常快，每天有新需求，每周都有新发布，每年都有大重构，每一次变化都有可能导致各种状况发生。越是贴近用户的系统，受下游服务影响就越大。因此，在项目中发生需求变更是难免的，我们不能抵制变更，但是一定要用切实有效的办法控制变更。变更控制的目的并不是控制变更的发生，而是对变更进行管理，确保变更有序进行。



需求发生变更后一定要认真仔细修改相关的文件，如需求文档、项目计划、设计文档等，并通过书面方式通知团队成员，不能在会议上一笔带过，并且要确保团队成员都准确地知道了变更的内容以及下一步的计划。注意，再小的变更都会给项目带来影响，多次微小的变更可能引发连锁变化，所以不能因为变更影响小而疏忽怠慢。

此外，公司的变更管理流程可以帮助我们有效控制变更，通过流程我们可以规范变更，并帮助我们将变更内容准确地传递给每一位团队成员。变更管理流程不是操作上的累赘，而是帮助我们控制项目范围、避免麻烦产生的利器。

3.3.3 开发进度管理



“向进度落后的项目中增加人手，只会使进度更加落后”，这句话摘自《人月神话[⊖]》。

伊格尔森定律（Eagleson’s Law）：“自己写的代码如果有半年时间没有看过，就跟别人写的代码没什么区别了。”这句话的意思是，代码看起来会很混乱、难以理解，并且同样无法通过进度估计来预测项目成功完成的时间。

在《华尔街日报》的一次采访中，微软 CEO 纳德拉透露自己每月会有一个周五和公司最高领导层开一次 8 小时的会，在另外的 3 周里开 4 小时的会，讨论的内容从财务表现到产品使用率的实时数据，覆盖面相当广。他这样做的目的是为了让公司的高级管理层能够保持统一战线。

为了有效控制进度，开发经理需要请各个小组每天召开“晨会”以检查昨天工作进展，并且布置当天工作；另外，每周五下午召开周例会，项目组全体成员都要参加。

1. 晨会

晨会的检查基准是各个小组的底层计划，以检查和确认为目的，不展开讨论。晨会规定在 15 分钟以内，每个小组的成员站在白板前面完成。步骤如下：

第一步，检查状态。成员逐个说明昨天任务的完成情况，今天计划的工作任务，以及遇到的问题。如果确认任务的完成情况不涉及执行细节，则仅需要回答：“完成”或者“未完成”。开发经理进行记录，一般任务按时完成可用打钩表示，未按时完成需要特别标记出来。检查完成后，将任务的完成情况分成三类：“按期完成”“延迟完成”和“延迟中”，并进行汇总统计。

第二步，调整计划。根据昨天任务的完成情况和个人任务的调整，小组一起对“底层计划”进行适当调整，确定当天每个人的工作任务。

第三步，解决问题。首先审核昨天列出的问题的状态。然后，将今天每个提

⊖ Fred Brooks 于 1975 出版的书籍，原名《 The Mythical Man-Month 》，被认为是软件管理艺术方面的一部权威著作。

出的问题记录到白板上。除非可以当场解决的简单问题，否则不对问题展开讨论，只记录到白板的问题栏。会后由相关成员讨论问题的解决方案，或另行安排时间组织专题讨论。

2. 周例会

周例会检查和调整项目计划，关注的问题是：任务完成了吗？没完成的原因是什么？怎么调整？先确认了状态，再讨论该如何调整工作或计划，并一定要落实到具体的行动方案上。

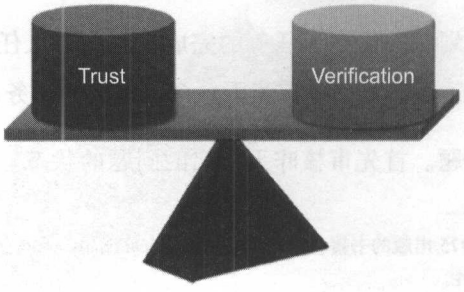
姓名		部门名称		项目名称		时间		上周主要工作			
编号	工作内容	周一	周二	周三	周四	周五	周六	周日	合计	结果	未完成原因
1											
2											
3											

只有提前建立工作结果的验收标准，才能确定如何才算是完成任务了。

项目计划要想落地，需要细化分解到每个人的工作任务中去。个人工作计划可以通过白板的方式管理，它可以实时反映进展情况。

计划进行调整和更新是常态，因为“计划本身没有用，不断地进行计划才有用”。底层计划的管理使用简单的白板就可以发挥巨大的作用，可见软件开发中“人”才是最重要的，没有“人”的主观能动性，再高级的管理工具也难以发挥作用。

3.3.4 评审的重要性



对于软件设计来说，评审与其技术设计方法本身是一样重要的，评审对于研发项目的成功而言是绝对必要的。对设计进行评审是为了尽早发现软件的欠缺，尽可能把这些缺陷在进入下一阶段工作之前予以纠正，从而避免后期付出更多的代价。

IT 系统在能够真正运行前，需要很多前期的分析、设计工作，这些工作决定了系统最终能否正常运行。但由于这些工作大多落在书面上，验证其正确性不够严谨，所以更多依靠一些有经验的专家通过书面审核的方式来确认工作是否达到要求，这就是我们常说的评审。

评审过程中涉及的角色主要有 4 种，包括责任人、主审人、评审专家、记录员。

3.3.5 软件版本管理

因为迭代的存在，软件会产生不同的版本。每个版本的软件都是一个完整的开发生命周期。软件的诞生可以认为是第一个版本，而版本升级的过程也就是软件不断长大的过程。

由于软件开发是一个独立的生命周期，可以与软件生命周期并行，因此自然就可以做到多个版本并行开发，以并行的方式来提高生产力。一旦同一个软件的不同版本生命周期并行地被推进，就需要把代码的生命周期单独切分出来，以确保不同版本软件工程师之间的代码在空间和时间上不会产生冲突。因为软件工程师编写代码是软件开发的核心生命周期，所以要确保他们的工作能够按照各自的时间顺序来执行编码生命周期活动。

多版本并发又会产生上线冲突的问题。线上版本的运行是软件生命周期的核心，不管有多少个开发生命周期并行，线上版本都只有一个，必须确保版本上线按顺序发生。另一方面，软件开发生命周期是软件生命周期中的非核心生命周期，不可以损害核心生命周期，所以必须要排队上线，并且要保证后续的版本要包含的前一个线上版本的所有内容，确保其连续性，这就形成了发布生命周期。

我刚开始工作的时候，我们使用的源代码版本控制软件是 CVS[⊖]，后来开始使用 SVN，再后来出现了 Git，我们来看看它们之间的差别。

Subversion（SVN）作为 CVS 的重写版和改进版，其目标就是作为一个更好的版本来控制软件，取代目前流行的 CVS。Subversion 的主要开发人员都是业界知名的 CVS 专家。Subversion 支持绝大部分的 CVS 功能 / 命令；命令风格和界面也与 CVS 非常接近。当然，不同的地方正是对 CVS 的改进。

从技术的角度来说，在 Subversion 中，“文件 foo.c 的第 5 版本”这个说法是错误的，正确的说法应该是：“文件 foo.c 在版本库被修改了 5 次，即执行 5 次 commit 后是什么样子”。显然，在 Subversion 中，版本库被修改 5 次后 foo.c 的内容和被修改了 6 次后 foo.c 的内容很可能完全一样，因为版本库的第 6 次修改很可能只修改了版本库的其他部分，而并没有对 foo.c 进行修改。相反，在 CVS 中，文件 foo.c 的第 1.1 版本和第 1.2 版本总是不同的。

CVS 只能对文件进行版本控制，不能对目录进行版本控制，因此 CVS 没有任何关于文件“移动”（move）操作的概念。当人为进行文件移动操作时，CVS 只能注意到一个文件在一个位置被删除了，而在一个新位置创建了另外一个文件。由于它不会连接两个操作，因此也很容易使文件历史轨迹丢失。设置 CVS 存储库时，必须非常谨慎地为每个文件选择准确的位置，因为在设置之后，几乎就要一直使用这个位置了。

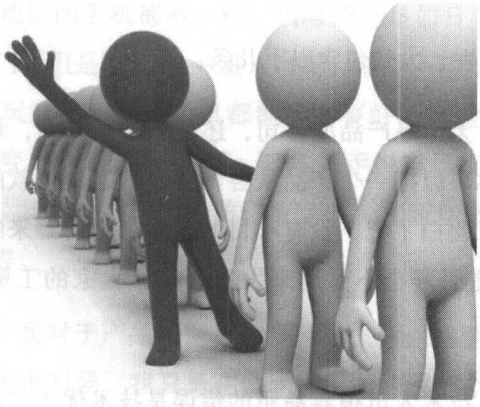
Subversion 将目录作为一类特殊的文件来处理（事实上，从文件系统的角度来看，目录确实是一类特殊的文件，当目录中的子目录 / 文件被删除、重命名、或新的子目录 / 文件被创建时，目录的内容将发生改变）。因此，Subversion 像记录普通文件的修改历史一样记录对目录的修改历史，当发生文件 / 目录的移动、重命名或拷贝操作时，Subversion 能够准确记录操作前后的历史联系。同样，像对文件的不同历史版本进行比较一样，Subversion 支持对目录的不同历史版本的

⊖ CVS 是一个 C/S 系统，是由多个开发人员通过一个中心版本控制系统来记录文件版本，从而达到保证文件同步的目的。

比较，可以清晰展现目录的变化历史。

与上面两者相比较，VSS 适合小团队使用，基本的配置管理功能都有。VSS 最大的特点就是部署比较简单，上手比较快。VSS 最大的缺点就是安全性问题、目录共享、文件方式存储等。当然 VSS 还只能在 Windows 下使用。

3.3.6　优先级安排



设置优先级的关键是，划去清单中那些既不紧急也不重要的事项，找到那些既紧急又重要的事项，并进行优先处理，然后再去做剩下的那些或是紧急或是重要的事项，这时候就需要你来安排合适的优先级了，优先处理既紧急又重要的事项是关键。我们常常把紧急的事情看作最重要的，最后反而耽误了解决重要的事情。

根据时间管理的经验，安排工作的原则顺序为：重要 / 紧急工作 > 重要 / 不紧急 > 不重要 / 紧急 > 不重要 / 不紧急。

记住，在资源和进度都有严格限制的情况下，需要考虑集中优势资源优先完成高优先级的需求。

3.3.7　项目管理重要性

项目管理是核心能力之一。其他方面能力的积累也往往需要依靠项目管理能

力的转化。譬如核心技术的积累，首先是基于最佳实践，而最佳实践只能依靠强有力的项目管理能力才能脱颖而出。而把最佳实践转化成解决方案，也是靠项目管理能力的支撑。最后再通过项目管理，把解决方案转化成核心技术产品或平台。

很多开发经理会看轻项目管理能力，其实这是错误的。项目管理是一种软实力，只有逻辑能力强、对于业务和技术都非常了解的人才能既做好开发经理，也做好项目经理，其实也只有两者都管理得当，我们的开发项目才能够真正发布成功。

项目管理的重要性，大致包含以下几条：

支撑公司收入：无论是产品型公司，还是外包型公司，最终都在通过一个个项目的形式落地，你可以把一个产品从咨询开始到客户付款的整个过程当成一个项目。这时候，大家明白了吧，没有项目的支撑，公司哪来的收入？哪来钱养活技术人员？公司的注册资金不是用来发工资的，大家的工资是通过客户付款产生的。

项目进度把控：技术人员很容易犯的错误是技术优先，对项目的进展不太关注，而项目管理的目标就是要让这一点不出现问题，也只有做好项目管理才能真正解决进度把控问题，你不能单纯地认为只需要给开发人员添加 KPI 就行了，如果真的可以，为什么会有 OKR 的出现？为什么日本的软件企业越来越不行了，要知道，他们可是 KPI 的发明人和强有力的执行者。

人员工作安排：缺少强有力项目管理能力的技术领导，很容易让团队的部分成员陷入迷茫。提出的命题很大，但是落实到每位技术人员身上，一部分人会迷茫，不知道每天应该干点什么，所以最好的方式是对工作进行拆分，每天的工作要能说清楚它的具体目标、要求标准等，这样才能让大家有存在感和参与感。

3.3.8 产品质量的重要性

乔布斯说过，苹果的产品从不做市场调查，因为如果问用户要什么，用户只

会在现有产品上提改进要求，不会提出创新的产品要求来。说白了，他们的理念是不问用户想要什么，问自己内心，问自己内心想要什么就好了。苹果要为那些疯狂到想要改变世界的人造工具，所以诞生了Mac、iPad、iPhone等经典产品，这确实是很牛的。他们的设计大师Ive说过，为什么用户喜欢苹果？因为用户在使用中会感受到他们打造极致体验的良苦用心。这种良苦用心确实是用户更喜闻乐见的。iPhone在2009年进入日本市场前，日本的手机是世界上独一份的存在，市场主流的Sony和夏普的手机，能看电视、能上网、能当电子钱包、能写邮件，这些属性和世界其他地区的手机都不一样。当时即使是如日中天的Nokia，也没有敲开日本市场的大门，但是iPhone一进入市场，短短一两年用户们就发现，不论是上网、写邮件、玩游戏，苹果手机都能给予更好的体验，随之日本手机市场的游戏规则发生改变，到了2010年后，日本的手机市场就再也不是原来的模样了。可见有些质量，比用户讲出来的质量更重要。直面自己内心，打造极致体验，也许是更高的境界。

刚刚工作的时候，我对于产品质量真的没什么概念。甚至，我对于什么是产品没有任何概念。因为那时候，我只是在做软件里面的一个小模块。完全不了解完整的产品是什么，更别提去想象客户会怎么用这个产品了。

质量是一种管理，是一种控制，是一种预防。只有成熟的组织，成熟的团队，才能做出高水平的产品来。

质量就是你的最高水平，以及你对用户理解的最高水平。你只有发挥出自己的最高水平，并且全身心投入，从根本上解决问题，才能带来更好的用户体验，更好的质量。

3.3.9 测试过程的区别

通用的测试过程包括计划、设计、实现、执行、完成几个步骤。

测试计划：需要确定这次测试目标和策略，估计测试用例、测试实现的工作量，确定所需的人力资源和测试环境资源。这些内容都写在测试计划中，测试计

划通过评审就可以执行了。

测试设计：需要确定测试需求，设计测试用例，并对测试用例进行评审等。

测试实现：任务包括搭建测试环境、编写测试脚本、编写驱动程序和准备测试数据。根据需要尝试测试部分程序，然后修改测试用例和驱动程序等。

测试执行：根据计划将测试任务分配给测试的执行人员，测试执行人员根据测试用例输入测试数据、记录测试结果。发现问题后记录和跟踪缺陷，缺陷修改完成后进行验证。执行中还要对测试环境进行管理和监控。

测试完成：主要工作完成以后需要对测试的情况进行分析、总结，确认目标是否达成，并给出测试结论或建议。具体的工作包括评估测试活动、分析测试结果、编写测试报告，最后对测试的整体情况进行评审并形成结论。

测试工作最容易被压缩，一旦进度紧张，开发经理就会本能地选择压缩测试时间。要知道，测试是保障交付质量最常用也是最有效的手段，V 模型中的每种测试都有自己的目的和针对性，不能相互替代。这好比在系统前拦上了一道道不同的网，只有每道网尽量多拦截缺陷，才能保证最终交付的质量，如果每道都放行，最后问题就会集中爆发。

单元测试与集成测试相比，测试对象有以下几点区别：

- 集成测试的被测对象是单元间的组合，不同模块往往是分配给不同的人员开发。集成测试主要关注不同单元模块之间的接口和配合度。
- 单元测试的测试对象是这些模块下实现具体功能的单元，一般是对应详细设计中所描述的设计内容。单元测试主要关注每个具体单元模块内部的逻辑结构和功能是否正确。
- 单元测试与系统测试相比，其侧重点在于发现程序设计或实现的逻辑错误，基本属于白盒测试的范畴。
- 单元测试使问题及早暴露，也便于问题的定位解决，单元测试属于早期测试，因而错误发现后就能明确知道是由哪一单元产生的。

- 单元测试允许多个被测单元的测试工作同时开展。

单元、集成、系统测试比较如下：

测试类型	测试对象	测试目的	测试依据	测试方法
单元测试	模块内部的程度错误	消除局部模块的逻辑和功能上的错误和缺陷	模块详细设计	大量采用白盒测试方法
集成测试	模块之间的组装和调用关系	找出与软件设计相关的程序结构、模块调用关系、模块间接口方面的问题	软件概要设计	结合使用白盒与黑盒测试方法，较多采用黑盒方法构造测试用例
系统测试	整个软件系统	对整个系统进行一系列的有效性测试	软件需求规格说明书	黑盒测试

3.3.10 测试驱动开发

测试驱动开发（Test-Driven Development，TDD[⊖]）是一种不同于传统软件开发流程的新型开发方法。它要求在编写某个功能的代码之前先编写测试代码，然后只编写测试能通过的功能代码，从而推动整个开发的进行。这有助于编写简洁可用和高质量的代码，并加速开发过程。

TDD 的基本思路就是通过测试来推动整个开发的进行。而测试驱动开发技术并不只是单纯的测试工作。

需求向来就是软件开发过程中感觉最不好明确描述且易变的东西。这里说的需求不只是指用户的需求，还包括对代码的使用需求。很多开发人员最害怕的就是后期还要修改某个类或者修改、扩展函数的接口，为什么会发生这样的事情？就是因为这部分代码的使用需求没有很好的描述。测试驱动开发就是通过编写测试用例，先考虑代码的使用需求（包括功能、过程、接口等），而且这个描述是无二义的，可执行验证的。

通过编写这部分代码的测试用例，对其功能的分解、使用过程、接口都进行了设计。而且这种从使用角度对代码的设计通常更符合后期开发的需求。可测试

⊖ Kent Beck 先生最早在其极限编程（XP）方法论中，向大家推荐“测试驱动”这一最佳实践，还专门撰写了《测试驱动开发》一书，详细说明如何实现。

的要求，对代码内聚性的提高和复用都非常有益，因此测试驱动开发也属于一种代码设计的过程。

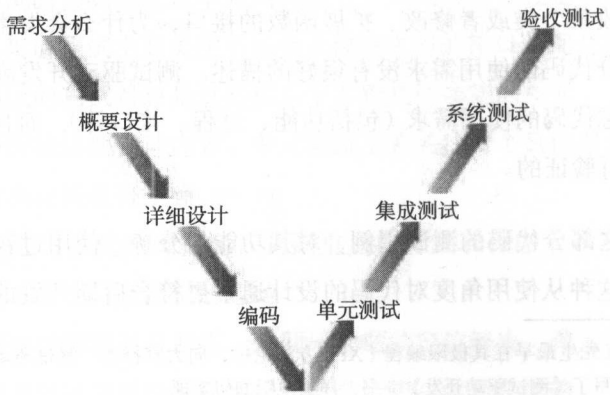
开发人员通常对编写文档非常厌烦，但要使用、理解别人的代码时通常又希望能有文档进行指导。而测试驱动开发过程中产生的测试用例代码就是对代码的最好解释。

1. TDD 模型

TDD 最重要的功能在于保障代码的正确性，能够迅速发现、定位 Bug。而迅速发现、定位 Bug 是很多开发人员的梦想。针对关键代码的测试集，以及不断完善的测试用例，这也为迅速发现、定位 Bug 提供了条件。

TDD 的基本思想就是在开发功能代码之前，先编写测试代码。也就是说在明确要开发某个功能后，首先要思考如何对这个功能进行测试，并完成测试代码的编写，然后编写相关的代码来满足这些测试用例，最后循环进行添加其他功能，直到完成全部功能的开发。

我们这里把这个技术的应用范围从代码编写扩展到整个开发过程。对整个开发过程的各个阶段进行测试驱动，首先需要思考如何对每一个阶段进行测试、验证、考核，并编写相关的测试文档，然后开始下一步工作，最后再验证相关的工作。比较流行的测试模型是 V 测试模型。



在开发的各个阶段，包括需求分析、概要设计、详细设计、编码过程中都应该考虑相对应的测试工作，完成相关测试用例的设计、测试方案、测试计划的编写。这里提到的开发阶段只是举例，需要根据实际的开发活动进行调整。相关的测试文档也不一定是非常详细复杂的文档或者什么形式，但应该养成测试驱动的习惯。

关于测试模型，还有X测试模型。这个测试模型，我认为，是对详细阶段和编码阶段进行建模，应该说更详细地描述了详细设计和编码阶段的开发行为，即针对某个功能进行对应的测试驱动开发。

2. TDD优势

相对于传统的结构化开发过程方法，它具有以下优势：

1）TDD根据客户需求编写测试用例，对功能的过程和接口都进行了设计，而且这种从使用者角度对代码进行的设计通常更符合后期开发的需求。因为关注用户反馈，可以及时响应需求变更，同时因为是从使用者角度出发的简单设计，也可以更快地适应变化。

2）出于易测试和测试独立性的要求，这将促使我们实现松耦合的设计，并更多地依赖于接口而非具体的类，最终提高系统的可扩展性和抗变性。而且TDD明显缩短了设计决策的反馈循环，使我们在几秒或几分钟之内就能获得反馈。

3）将测试工作提到编码之前，并频繁地运行所有测试，这样可以尽量地避免和尽早发现错误，极大地降低后续测试及修复的成本，也提高了代码的质量。在测试的保护下，不断重构代码，以消除重复设计，优化设计结构，提高代码的重用性，从而提高软件产品的质量。

4）TDD提供了持续的回归测试，使我们拥有重构的勇气，因为若代码的改动导致系统其他部分产生任何异常，测试都会立刻通知我们。完整的测试会帮助我们持续地跟踪整个系统的运行状态，因此我们就不需要担心会产生什么不可预知的副作用了。

5）TDD所产生的单元测试代码就是最完美的开发者文档，它们展示了所有

的 API 该如何使用以及是如何运作的，而且它们与工作代码保持同步，永远是最新的。

6）TDD 可以减轻压力、降低忧虑，提高我们对代码的信心，使我们拥有重构的勇气，这些都是快乐工作的重要前提。

7）提高了开发效率。

3. TDD 的重要性

（1）反映真实需求

这里存在先写测试和后写测试的区别。

先说后写测试。根据很多经验，在直接写产品实现代码时，需要考虑需求与实现的细节，往往很容易因一时疏忽做不到两者兼顾。

有人会说我可以通过后写测试来保证。第一经验是很多人都不会在实现完成后补充测试，因为还有更多的工作和需求需要实现。第二是开发人员很容易在后补的测试中只试图测试已有的实现，而不是需求本身，很容易遗漏掉一些边界检查之类，在测试时，已有的实现细节会在脑子里面先入为主，即使实现存在问题或漏洞，也很难在后补的测试中测出来。我阅读过很多面试者的测试代码，很明显都是后补的，因为一些很明显的问题没有测出来，甚至已经实现的逻辑也是只测试了一部分。而这些面试者也都承认。这样的结果就是，一旦代码嵌入开始集成之后就会暴露出问题，而后补的测试起不到对实现代码的保障需求作用，开发者不得不借助调试工具，单步跟踪实现代码的每一行去寻找问题发生的原因。

再说先写测试。按照 TDD 的流程，先写失败的测试，再写恰好让测试成功的实现代码，最后重构，如此往复。这样的好处在于：每先写一个测试都是在试图用自己对问题和需求的理解来定义实现代码的框架。从测试的不同角度，范围、边界、大小、功能等来定义实现代码将来会是个什么样子。一个测试接着一个测试，利用 TDD 的过程，恰好不多不少，正好驱动出解决这个需求和问题的实现代码来。

这里的重点在于先写测试可以让开发者把重点放在理解需求和实现需求上，

而不是一开始就陷入实现的细节中，掉入两者都兼顾不好的境地。先写的测试代码，作为副产品，可以作为验证需求的得力保障。

（2）设计在其中

先写测试对于设计的好处在于，先写测试会先定义新的类，以及定义类与类之间的关系，就是定义类与类之间如何交互，每个类如何暴露自己的接口，类和类之间的引用关系。这时，开发者会认真考虑如何分解类与类之间的耦合关系，这样产生的实现代码利用了 IoC 和 DIP 的模式，会更容易实现面向接口编程。

先写测试的好处还在于，代码的可测试性很高，在加入更多的测试代码和新类的时候，同样借力于已有类的面向接口和依赖反转所带来的可测试性，从而达到新实现代码的面向接口和可测试性，形成良性循环。而这对于整体的代码和设计将十分有利。换句话说，测试即设计。

回头看后写测试的情况，因为从一开始开发者就把重心放在实现的细节和功能需求的往复上，对于代码设计、类的关系和定义很容易疏于考虑，造成耦合紧，可测试性差的情况。

（3）增强信心

在我看来，软件开发周期、软件交付最大的问题在于交付后的运行和维护阶段，这两个阶段才是软件在持续交付价值的时期。软件的可维护性，在这个阶段凸显价值。在软件交付后，包括软件开发周期期间，不可避免的就是开发者再根据新的需求，逐渐添加新的功能代码，或者修复一些已知的缺陷。

很多经验表明，在开发者按照需求添加一些新的代码进入系统，或者试图修复已有缺陷时，很容易导致既有功能出错，也就是新引入的代码打破了既有代码的逻辑，从而导致回归问题的出现。因为软件系统的可测试性差，无法做到快速、频繁、自动的回归测试，带来的可维护性自然也很差。

而作为 TDD 的副产品之一——可以快速频繁自动运行的测试代码，可以在开发者新引入代码之际，给予开发者足够的信心，每次添加一点新代码，一个方

法，一个类，都需频繁运行已有相关的测试代码，来确保新引入代码不会打破已有的功能。

在持续集成中，这些测试代码可以帮助验证每次嵌入的代码都不会破坏已有的功能。这也是《重构》里面反复提到的在每个重构小步骤后都要运行所有的测试代码的原因所在。

（4）粒度和进度

按照 TDD 的原则，先写测试，可以让开发者在同一时间只关注功能需求的一小部分，把功能需求细分到一定小的粒度，用测试代码去表示这样的需求，用实现代码让测试通过，实现需求，然后重构。

这样的好处在于开发者自己的注意力和重心不用在整个功能需求内的小需求点之间犹豫，每次注重解决单个小问题，解决完一个再进入下一个小问题的解决。在保证小粒度实现的同时，保证进度即使被打断时，已经做完的内容是完整可以运行的，即至少是实现了部分需求的代码。

而后写测试的代价是实现代码很有可能包含设计上的问题，甚至含有缺陷，在完美的测试代码完成之前（事实上这是不可能的）可以交付的是可能存在严重缺陷，甚至是曲解了功能需求的实现代码。

4. TDD 总结

TDD 能够带来的好处：

- 减少开发周期中的反馈；
- 提高代码质量；
- 保证设计质量；
- 集中精力开发一个功能。

TDD 能改善和验证设计：

- 以客户端的视角编写测试代码；

- 为客户端提供示例代码；
- 更注重接口的设计；
- 为了便于测试，需要实现松耦合；
- 更少的调试时间。

3.3.11　自动化部署工具介绍

随着敏捷开发和 DevOps 的流行，自动化部署成为软件企业的关键能力，决定了企业的生产力水平。在网站、公共服务和云计算领域，部署是把开发人员的交付件安装到测试环境、类生产环境和生产环境等，是 DevOps 中的重要组成部分，决定了软件的交付效率和质量水平。手工部署费时、低效也无法保障质量，在对外提供多节点集群服务的互联网公司，手工部署变成了噩梦。而自动化部署的目标就是要取消手工操作，将全部步骤流程化、标准化、规范化，从而提高质量和效率。

下面介绍几种常用的自动化部署工具。

1. Chef

Chef 是一款自动化服务器配置管理工具，可以对所管理的对象实行自动化配置，如系统管理，软件安装，开发语言是 Ruby。Chef 由 Chef Server、Chef Workstation 和 Chef Node 组成。Chef Node 是安装了 Chef-client 并注册了的被管理节点，Chef-client 连到 Chef Server 取得最新的配置指令（Cookbook）并按照指令配置自己。Chef 的优点是有丰富的模块和配置脚本，缺点是需要熟悉 Ruby，学习曲线比较陡峭。

2. Puppet

Puppet 是一个开源的软件自动化配置和部署工具，开发语言是 Ruby，支持 Linux、UNIX、windows 平台，使用自有的 Puppet 描述语言，可管理配置文件、用户、cron 任务、软件包、系统服务等（Puppet 把这些系统实体称之为资源）。Puppet 的设计目标是简化对这些资源的管理以及妥善处理资源间的依赖关系。

Puppet 采用 C/S 星状的结构，所有的客户端和一个或多个服务器进行交互。每个客户端周期（默认半个小时）向服务器发送请求，获得其最新的配置信息，并且保证和该配置信息同步。每个 Puppet 客户端每半小时（可以设置）连接一次服务器端，下载最新的配置文件，并且可以严格按照配置文件来配置客户端。配置完成以后，Puppet 客户端可以反馈给服务器端一条消息。如果出错，也会给服务器端反馈一条消息。

Puppet 的优点是社区支持好，有成熟的接口，能够支持每一种操作系统，安装配置简单，并且有很强的报表能力。它的缺点是你仍然需要熟悉 Ruby 和命令行 CLI。

3. SaltStack

SaltStack 是一个服务器基础架构集中化管理平台，允许管理员对多个操作系统创建一个一致的管理系统，包括 VMware vSphere 环境。SaltStack 也属于主从（C/S）结构，由主控端（master）和被控端（minion）基于证书认证。SaltStack 与特定的命令结合使用可以在一个或多个 minion 执行，它具备配置管理、远程执行、监控等功能。SaltStack 基于 Python 语言实现，结合轻量级消息队列（ZeroMQ）与 Python 第三方模块（Pyzmq、PyCrypto、Pyjinjia2、python-msgpack 和 PyYAML 等）一起构建。

SaltStack 的优点是社区活跃，输入输出和配置采用 YAML 且格式清晰一致，扩展性和弹性很好。它的缺点是新手安装上手不易，Web 界面没有竞争力，非 Linux 系统支持不是很好。

4. Ansible

Ansible 是新出现的自动化运维工具，基于 Python 开发，集合了众多运维工具（Puppet、Cfengine、Chef、Func、Fabric）的优点，实现了批量系统配置、批量程序部署、批量运行命令等功能。Ansible 基于 Python 开发，具有众多的模块，可以提供幂等性能操作。Ansible 有以下优点：

1）相比其他自动化集群管理和运维工具 Puppet、Chef，Ansible 显得极其简单并且轻量级，也可以和 Puppet 一样进行模块扩展。

2）轻量级的好处是学习门槛低、问题少、安装快、执行快。操作完全依赖 SSH 而不需要安装 Agent。这样的好处是不再需要维护 Agent 的状态，不用担心 Agent 挂掉。而 SSH 是每台服务器必备的服务。

3.3.12　编写高质量代码的难度

任何程序员都能写出机器可以阅读的代码，但只有好的程序员才能写出人可以阅读的代码。这句话道出了要写出容易阅读的代码的难度。

以程序命名为例。我们在程序代码中，往往看到很多类似 icount、var、num 这样名字变量，还有很多叫作 manager、controllor 的类，这些都是因为我们想不到应该如何命名导致的。除了名词匮乏，我们的动词往往也很匮乏，证明就是：我们的很多函数名都叫 Process()、Run()、Poll()、Loop() 诸如此类。如果我们把程序源代码看成是一篇文章，那么这篇文章的词汇，就是各种变量和函数的名字。如果我们在命名上困难重重，那么这篇文章也一定晦涩难懂。

命名上的困难，除了因为我们英语词汇量太小以外，另外一个原因是我们对于业务领域的不了解。在接到需求后，我们往往就急着开始所谓的设计和开发。如果我们把程序仅仅看成一些数据和算法组合，那么我们的命名必然也只是局限于这些数据结构、算法的概念上，比如我们常见到 xxMap 的结构体，还有 xxCallback 的函数指针。用这样一堆名字构建起来的程序，就好像摩斯电码一样难以理解。尽管在这些看起来都差不多的字符背后，实现的是一个个鲜活而独特的业务需求，但是光看字面是完全无法想象出来的。这个问题实际上也很好解决，就是我们在写程序之前，多去了解这个程序所在的应用领域，看看这个应用领域里面到底是有些什么样的词汇。

比如一个商业应用中，就会有 Bill、Invoice、Deal 等专用词汇，在游戏应用中，有 Player、NPC、Monster 这些概念……我们可以在几乎任何时候，都能从

这些业务领域中攫取大量的词汇，来替换掉计算机领域中少得可怜的几个词。如果我们真正把代码中的命名，变成应用领域的词汇，那么这样的代码片段，就是一个描述某种业务领域的文章，如此，可读性就能大大加强。

命名本身并不影响程序的运行，但是我们也没必要直接写出好像被扰码器处理过一样的代码。如果我们的命名词汇既准确又生动，那么我们的代码一定也是非常容易读懂的。特别是，我们阅读代码的目的常常不是要评估代码的算法，而是找到某段业务逻辑的位置来进行修改，一个和业务逻辑有关联的命名，能让我们快速跳过大量不相干的代码，直接定位到需要修改的地方，这对代码维护是非常有利的。

我们知道，面向对象编程，需要以对象类型来对业务代码进行建模，而由于汉语名词的匮乏，我们常常在表达一个对象时，找不到一个专有名词来表达，而是用“做什么什么的东西”来表达这个对象，这对我们代码的设计会造成极大的困扰。因为行为的特征在对象上往往是不够稳定的，一旦我们以行为作对象的名字，而这个对象在后续的迭代中屡次被修改，这样就很容易出现名不符实的情况。

因此我们在设计面向对象代码的时候，不能仅仅以汉语的习惯去设计，而是要多找找有没有专门表达这个对象的英语名词。

如果我们想写出如同自然语言一样易读的软件代码，那么就一定要用自然语言来写文章的结构。但是很可惜的是，自然语言的文章以传情达意为目的，而软件代码主要是控制电脑工作的任务列表。这两者之间一个重要的差异就在于“语句”的存在形式上。

我们知道，代码是一行行执行的，而电脑对于数据的处理，往往存在很多类似的、重复的处理步骤。此时，就用到了“封装”：我们把类似的、重复的代码封装成子函数；用继承的方法来构建相似的数据对象。如果我们还能用恰如其分的名字来命名这些子函数和子类型，那么我们就能避免长篇累牍的重复代码，从而让代码更容易理解。也许，这种封装是一种“额外”的劳动，因为 CPU 不管这

些，如果你只是想算出结果，那么完全可以用一个函数从头写到尾。但是如果你有意识地做一些有具体业务含义的封装，你会得到另外一个好处就是代码能更方便被重用。代码重用的首要条件是代码可理解，而封装正是对复杂的实现过程进行屏蔽，让人可以快速理解。而业务领域的重复逻辑是非常常见的，如果代码刚好是一些典型的业务流程，那么这些对应流程的代码，就一定能被重用到大量的类似业务流程的处理环节，这样的好处不言而喻。

3.3.13　业务代码对技术的作用

真正的技术和业务驱动的差距在于业务看的是今天，或者是短暂的明天，而技术放眼的是未来。但这并不是说学习算法和基础没有用，这是为业务提供理论基础，只有将技术应用到业务中，才能充分发挥它的作用。

大家要明白，技术只有支撑了业务，你才有未来，不能解决当前业务问题的技术人员，不要谈理想，因为只有能够解决问题的人，才配谈技术理想。当业务出现问题的时候，技术人员应该立即站出来，去了解、理解业务，用自己的技术手段结合业务来解决问题，而不是觉得学习技术没有用，这种想法本质上是错误的。

3.3.14　迭代的意义

软件无法一次性把所有的业务都模拟出来，必须要分步骤将一个一个阶段做出来，通过用户的反馈，一点点地升级。这就是所谓的迭代，每一个小的迭代都是瀑布式的推进，每一个迭代对下一个迭代也是瀑布式的推进，整体仍然是瀑布式的推进。所以瀑布式的推进，也就是活动按时间顺序发生，本身是没有问题的。

迭代应该如何进行呢？迭代的前提是必须要先确定优先级，而理解清楚业务的核心生命周期是最高优先级。首先要实现业务的核心生命周期，一旦业务的核心生命周期模拟出来之后，业务人员就可以操作软件了，进而可以对软件进行反

馈。形成反馈环后，软件就可以通过一次一次的迭代，丰富核心生命周期的功能，或者进行架构拆分，形成新的非核心生命周期。

一个迭代会根据业务需求组织相应的资源。这些资源以人为主形成组织架构，这个组织一般被称为项目。对于一个软件而言，就会有多个迭代，也就是多个项目并行进行。因此需要项目经理收集每个项目，也就是每个迭代的进行状况，对不同项目作出不同的调整。

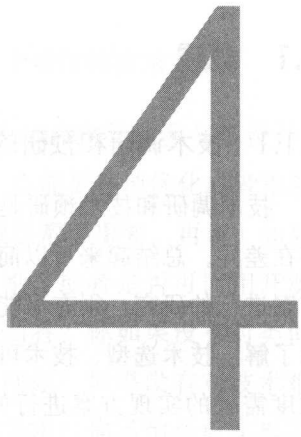
3.3.15 倾听的重要性

当你听别人说话时，你真的听懂了他说话的意思了吗？你是不是也习惯性地用自己的想法打断对方的话？不听人把话说完就断章取义下了定论，误会了对方的美好意愿，扼杀对方的创造力和主动性。

我们以用户需求和产品需求差异为例。由于立场、关注点不同，用户需求提出方和产品需求制定方很容易发生分歧。要想确保用户方满意，必须从用户的角度看问题，了解清楚用户表面需求背后的“需求”，换位到用户的立场，如果你自己作为用户都不满意做出来的产品，那你的产品绝对是有问题的，晚改不如一开始就充分理解清楚，避免出现分歧，因为最终是由需求提出方（产品经理）来判断项目成败。

3.3.16 事必躬亲的错误

事必躬亲是对员工智慧的扼杀，长此以往，员工容易形成惰性，责任心大大降低、技术能力止步不前。情况严重者，会导致员工产生腻烦心理，即便工作出现错误也不愿意向管理者提出。多让员工参与到需求讨论、软件设计中，这是对他们的肯定，也是满足员工自我价值实现的精神需要。赋予员工更多的责任和权力，他们会取得让你意想不到的成绩。



第 4 章

技术调研 / 预研

技术方向需要通过技术调研、技术预研来帮助团队成员逐步认识、理解和明确，这是一个必然的过程，技术团队领导应充分重视并直接参与。

本章主要介绍和解决以下问题：

- 为什么需要进行技术调研。
- 什么是技术调研，具体的思路、过程是什么样子的。
- 什么是技术预研，具体的思路、过程是什么样子的。
- 其他相关讨论。

4.1 概述

4.1.1 技术调研和预研的差别

技术调研和技术预研是存在一定差别的，主要是立项时所处的环境、目的等存在差异，总结起来可以简单用一段文字描述：技术调研是针对粗粒度需求实现方案进行的研究，很有可能对所需技术根本不清楚，需要通过调研项目来完成技术了解、技术选型、技术可行性分析、技术方案设计等工作。技术预研是针对细粒度需求的实现方案进行的预先尝试，主要针对的是通过技术调研所选择的技术，同时结合我们产品化时的实际需求，对实现时存在的不确定性因素、细节等进行预先研究、尝试，从而减小产品化过程中的技术实现风险。

正是由于所处环境、项目目的的不一致，由此产生的技术调研报告和技术预研报告的内容也会差别较大。技术调研报告首先阐述从需求整理到明确方向再到搜索技术调研方案及初步筛选的过程及结果，接着介绍具体的测试方案确定、测试环境搭建、测试用例编写及运行的过程，最后针对调研项目中得到的一些实际业务场景下的运行数据进行对比、分析，通过解释数据产生的原因来明确下一步计划。技术预研首先根据细化的需求（有可能是产品化需求的一部分难点需求）对系统整体进行定义，接着通过列举方案明确本次预研论证过程，再通过论证过程反复执行列举、论证、推翻、再列举这一过程，最后是对本次预研获得的最终结论进行总结。

从最终的输出报告来看，技术调研一般针对每一项技术都会有一份单独的报告，然后汇总成一份调研报告（对各项进行汇总、对比、总结），而技术预研一般只有一份报告，通过这份预研报告说明预研项是否成功，是否可以采用该种方案、技术并进行下一步的产品化立项。

4.1.2 参与调研和预研的重要性

对于一个技术负责人来说，职责范围内有许多工作，比如组建团队、了解产品，但更重要的是设计靠谱的技术方案。故首先要了解系统存在的问题、产品未

来的走向和技术团队的现状，针对这三点，亲自设计一个最优的技术方案。

为什么技术负责人要亲自设计呢？因为技术负责人亲自主导或参与了设计，就能有针对性地去解决问题，即便将来系统遇到瓶颈，也能更好地优化或者重新设计。不要用各种理由来回避做这个事情，在任何阶段这都很重要。再者，如果别人问你，你的设计和其他公司的设计方案有什么区别，或者是否可以用开源框架搭建起来从而替换你们自主研发的框架，你怎么回答？你如果没有对类似的技术、框架进行过调研，怎么能够准确地回答这些问题？如果没有对技术难点进行预研，你怎么敢把设计对应的需求写入产品化需求？你不怕最后实现不了吗？

为了更好地设计系统、理解技术，你一定要组织调研项目和预研项目，这是因为你或者团队不可能什么技术、框架都懂，更何况技术发展非常快，只有针对技术进行调研、预研，才能够真正跟上技术发展的潮流，否则终有一天你会被技术岗位淘汰。

4.2 技术调研

4.2.1 技术调研思路

到底怎样做才算是完整且优秀地完成一次技术调研？

教技术的书籍很多，但是教调研的书籍几乎没有，即使有也不会教那么细。我曾因这类工作而彷徨、受挫，现在又看着新人彷徨、受挫，于是就有了想法进行尝试总结一个范式出来。我觉得想要做好一个技术调研项目，要了解需求、挑选技术，还要进行技术可行性分析、测试、输出分析结果。

1. 了解动机

除了自己发起的技术调研，其他技术调研也都需要先了解需求。“了解需求”这是一个人尽皆知且每个人在技术调研前都会去做的一件事。但不夸张地说，在

这个阶段栽跟头的人最多。

很多人，特别是新人，在这个阶段出问题的原因大概有以下几点：

- 畏畏缩缩，担心一开始问太多会显得自己很无知，担心对方轻视自己；
- 听到几个关键字就以为已经了解需求，没有在意对方说的一些细节；
- 在对需求有疑惑的情况下还硬着头皮做，缺乏沟通意识；
- 没有分阶段与需求方沟通，可能在快完成时才发现需求理解错误要推倒重做。

收到调研需求之后，我们需要整理并明确整个调研项目的背景。为什么会有这个技术调研项目存在？这一点肯定是有答案的，例如不满足用户的高并发需求、不满足用户对于各种 Chrome 浏览器的支持。明确了用户背景，就很容易描述该项目提出的背景了。

2. 明确目的

了解了调研动机之后，接下来，需要明确本次调研项目可能涉及的具体技术内容（可以不明确，通过模糊搜索方式进行初步筛选即可），确定调研过程中需要对各个技术的差异点、技术实现原理进行总结，并通过测试数据、数据对比及原因分析，分析哪种技术或者框架适用于用户提出的实际需求。这些都是调研的目的。建议你在这一环节增加提问方案，自己给自己提出一系列问题。

调研的过程是怎么样的、为什么选择这些技术作为调研方向？

1）这些技术分别是什么？
2）技术调研方案、用例分别是什么？
3）各个技术或框架之间的区别是什么？
4）各个技术在通用场景下的优缺点是什么？
5）各个技术在用户提出的特定业务场景下的优缺点是什么？
6）根据得出的测试数据，为什么会有这样的结果？

针对这些问题，我们可以进一步明确在本次调研过程中，需要获得以下数据

和内容，才能回答以上这些问题：

1）总结各个技术或框架的实现方案、细节、原理；
2）记录调研过程中的各个细节；
3）明确调研测试方案、测试用例；
4）列举各个技术之间的区别；
5）记录测试数据并从原理上解释；
6）列举各个技术是否适用于业务场景；
7）为技术预研项目输出调研结论。

3. 确定步骤

在充分了解需求的前提下，调研本身会显得轻松点。

需要注意的是，进行调研时要合理安排时间，调研过程往往伴随着对新知识的探索，很容易“沉迷于学习”。别忘了这是一项工作。

个人有个小技巧，按照以下步骤来做往往效果不错：

1）尽量多地收集各种方案和资料；
2）迅速粗略地过一遍上述方案和资料，大体上总结出几种可能合适的方案；
3）针对几种方案，一边调研每种方案，一边做笔记；
4）最后拿着笔记做最后的横向对比；
5）得出结论，同时因为做了笔记，反馈的素材也有了。

以上是关于“如何做”。需要说明的是，这只是我的个人习惯，你有自己的做事风格更好，没必要强行一致。

还有一点需要注意的是，千万不要埋头苦干。“沟通”应该是贯穿始终的一件事，前面也提到了，对需求的理解偏差可能会导致整个调研工作推倒重来。那么该如何沟通，以及沟通些什么呢？

对于第一个问题，如何沟通，我的方案是，阶段性地去跟需求方或者跟有经

验的同事讨论。比如一个技术调研分为四个阶段，那每完成一个小阶段，就可以尝试去沟通一次（必须强调一下，规则是死的人是活的。假如对方很忙的情况下，你偏要强行打扰对方去沟通，或者一个很小的技术调研你也按阶段多次去沟通，就尴尬了）。

对于第二个问题，沟通的内容，我认为主要有以下几点：

1）对需求细节分别进行确认；

2）将自己的工作进度汇报给对方（这一点很重要，一方面是让对方知道你在做什么及完成到哪个阶段，另一方面是假如你路走偏了，对方能及时知道并纠正）；

3）将自己当前的工作成果告知对方。

一般来说，一个调研过程分为以下几步，这些步骤也应该体现在你的调研报告里：

1）调研过程介绍：对需求进行整理，明确调研的方向，初步筛选，选择技术调研过程；

2）调研技术简介：逐一说明调研技术的实现原理、规则；

3）测试环境：对测试环境进行说明；

4）调研技术测试方案：包括通用测试方案和业务测试方案；

5）调研技术测试用例：包括通用测试用例和业务测试用例；

6）调研技术测试数据对比 / 分析：包括对通用测试场景和业务测试场景的测试结果数据进行对比并分析原因；

7）调研总结：对业务方案进行总结，从性能、技术评测、需求方、产品化、综合等多个不同的角度评判调研结果，选择可以用于技术预研的技术；

8）列举现有不足和下一步工作计划。

注意，需要明确的是，针对每一种技术或框架，调研过程中应该明确采用相同的测试场景、测试数据、测试方案及用例，绝不可以有差异，否则调研结果本身就会受到质疑。

4. 结论跟踪

做完技术调研后，一定要有成果。这里所说的结果可以是调研之后发现“某个方案是最佳的”，也可以是调研之后发现“尚无解决方案”，还可以是调研后对需求本身提出质疑。

反馈有几种常见的展现形式：

- 比较大的技术调研可用 PPT 的形式展现出来。比如有同事调研“兼容 Android 6.0 权限管理”，用一个 PPT 将技术方案的选择、6.0 权限管理的原理、最终方案的选取等内容分享出来就特别好。
- 假如是简单的技术调研可以以文档的形式展现，推荐用 markdown 来写，也可以 github/gitlab 直接展示。
- 可以以邮件或口头的形式进行反馈。

个人比较推荐文档的形式，因为这适合大部分调研工作。反馈的内容有几点是需要写进去：

- 简要说明下调研需求；
- 介绍跟需求相关的前置知识；
- 目前有哪些方案，具体分析各个方案的优缺点及适合的场景；
- 技术调研的结果是怎样的，不可行的话是因为什么，可行的话说说最终决定使用何种方案（自己无法决定的话可以开个分享讨论会），并说说该方案跟其他方案比有何优势；
- 假如是新库的引进，需要简要介绍该库的使用方法及内部原理；
- 调研过程中碰到了哪些问题，如何解决；

总而言之，把一次技术调研当成一次绝佳的学习机会来做，那反馈的内容就不会显得空洞。

反馈的时机，需要在保证质量的前提下，尽量主动、提前向需求方或组内其他同事提出。一方面是你的反馈对别人而言也是一个学习机会，另一方面主动推送一件事是一种优秀的表现。

4.2.2 调研过程

一般来说，调研过程需要按照“需求整理→明确技术方向→搜索技术调研方向→调研方向初步筛选→选择技术调研方向→确定调研测试方案及用例→调研执行→调研数据对比→调研结论”这样的顺序，当然调研的执行过程也可以并行执行。

1. 需求整理

我们首先需要明确这次调研背后实际的业务需求，比如说需要支持什么样的业务场景、是否需要支持横向扩展、业务的实际场景是怎么样的、是否包含高可用的海量数据读取等难度较高的技术、是否需要我们使用开源技术（因为可以看到源代码，闭源或者商用版本容易出现后续维护和费用问题），这些都是业务方会提出的需求。

我们需要将上述业务需求转化为技术需求，从技术层面理解业务需求，例如高可用就可以转化为“没有单点失效，任何一个单点都不会造成数据不可访问”，又比如是否需要支持横向扩展，可以转化为“数据动态分散在不同的服务器上，可以通过动态添加服务器节点增加系统容量”。

技术需求转化完成后，我们可以明确本次调研项目的技术调研方向。因为只有明确了技术方向，我们才可以进行技术选型，进而继续我们的调研工作。

注意，业务需求转化为技术需求后，最好可以组织一场内部的评审，喊上产品经理、技术骨干，一起讨论是否存在理解偏差。

2. 技术选型

不知道你有没有遇到过这种场景：开发团队内有人不断高呼某某新技术、框架，想把最新、最热的技术应用到项目里。这些人可能是因为读到了相关的博客，或者是看到了 Twitter 上的讨论，也可能是刚刚在 InfoQ 的技术大会上听到了精彩演讲。开发团队开始采用这种时髦的新技术（或者软件架构设计范式），结

果他们却没法更快开发出更优秀的产品，反而开发的速度降下来了，后续版本的交付也出问题了。有些团队甚至干脆专心修复 Bug，停止开发新功能。这是为什么？这是由于没有根据实际需求进行技术选型。

一般来说，技术选型的决定方式有以下三种：

1）有的团队会根据社交媒体上的讨论来决定选择哪种架构；

2）团队会跟风走，哪个热门就选哪个；

3）有的团队坚持通过技术评估手段进行选型。

前两种方法没有评估在实际需求驱动下的性能数据，属于简单粗暴的方式，一定会为未来埋下隐患。第三种方式首先会明确该技术 / 框架所使用的需求范围，根据需求明确技术调研方向，然后通过技术评估手段进行选型，最终的选型结果需要需求提出方、技术调研方、技术专家代表等多个维度人员参与，以事实作为评判标准。

（1）先测试、研究，再决定

针对新技术提供的功能，在决定采用之前花一两天搭个原型，然后组织大家分析利弊。你可能会遇到若干能彼此替代的技术，可以让团队里不同人用不同的技术来搭原型。

（2）何时开始

原则上说，应当选择投资回报最大的时间点开始。大多数技术是用来解决特定问题的。你遇到了那个问题吗？那个问题重要不重要？会不会节省很多时间？新技术带来的好处能不能抵消学习成本和重做的成本？如果我们的开发速度从一开始就降低到正常水平 1/2 甚至 1/4 会如何？想想新技术还值得吗？

优秀的团队有更多自主权——一些团队确实比其他团队更快出成果，他们也更容易厌烦自己手头的工作。这些团队可以更多、更快地引入新技术，但这不是省略快速搭建原型或者黑客马拉松的理由。相反，如果这样的团队在交付上遇到了麻烦，一定要加倍小心。

（3）找到对的人

有良好技术背景的人——那些人了解不同的范式，理解编程的理论（算法和并发），受过良好工程文化熏陶，这样的人很少会去凑热闹。

有经验的人——年轻的开发人员更喜欢凑热闹。如果有多年的开发经验，见过许多技术，踩过许多坑，在技术决策时就更容易做出客观的判断。

（4）技术采用生命周期

技术采用生命周期，即 Technology Adoption LifeCycle，是一个用来衡量用户对某项新技术接受程度的模型。这个理论最早源于 1943 年 Ryan 和 Gross 对玉米新品种的扩散行为研究，而后 Everett M.Rogers 提出了扩散曲线，并将采用者分为五种类型。

技术采用生命周期的形状是一个钟形曲线。这一曲线将消费者采用新技术的过程分为五个阶段，分别包括创新者阶段（2.5%）、早期采用者阶段（13.5%）、早期大众阶段（34%）、晚期大众阶段（34%）与落后者阶段（16%）。

- 创新者阶段：有很多的坑，但却不乏众多爱好者乐于踩坑。
- 早期采用者阶段：有一些开发人员在尊重自己的直觉和喜欢的前提下，开始使用该技术。
- 早期大众阶段：各大论坛、技术会议开始纷纷报道，这时候需要慎重思考之后决定是否采用。
- 晚期大众阶段：标准、社区已经完善，大部分人都在使用了。
- 落后者阶段：除非万不得已，否则没人再用了。

我们的使命是使用创新技术并将它们应用在我们的软件开发过程中，并且能够解决新技术所带来的应用鸿沟，因此，处于创新者阶段、早期采用者阶段的技术不适合我们，因为它还没有形成完整的社区支持，已经处于落后者阶段的技术也不适合我们，因为我们已经有更多的选择了。

（5）何种方式

可以通过网络搜索（进入 Google，搜索你选定的关键词，最好是若干个），找到

国外专家推荐的相关技术，逐一记录。接下来，按照技术方向进行分类（很多技术或者框架的原理都是类似的，可以归为一类），然后按照某种条件进行筛选，例如按照开源协议的约束条件、宽松程度进行重点考虑，或者根据社区的热度进行筛选。

3. 明确方案

经过前一步的挑选后，我们明确了技术选型，接下来，需要明确实现方案和测试方案。

（1）实现方案

实现分为设计、编码两个步骤，首先需要针对通用场景和业务场景分别进行设计。通用场景一般是针对几个技术的通用能力进行比较。以数据库为例，我们需要测试几种不同数据库的读、写能力，包括串行读 / 写和高并发读 / 写，这两点都需要进行测试程序的设计和编码。

（2）测试方案

测试场景分为通用测试场景和业务测试场景两部分，虽然分为两个测试场景，但是测试方案一般来说是一样的，差异点是执行场景、并发程度不同。还是以上面说的数据库测试场景为例，无论哪种场景，我们都需要通过执行 SQL 语句或者类似数据“增删改查”的方式进行测试，如果是关系型数据库，DML 语句一般会涉及 SELECT、INSERT、UPDATE 等。

接下来是针对通用测试场景和业务测试场景分别设计测试方案。通用测试场景指的是针对数据库本身进行高强度的插入、更新、读取等单一操作测试，不需要按照实际业务场景的数据量，可以任意扩展，即通用测试场景不会对插入数据量、读取数据量设置上限。这里需要注意，数据表最好和业务场景一致。这一方案积累的数据可以作为通用场景下的对比基准数据，为后续调研报告提供支撑。

业务测试场景指的是完全模拟现有或者未来需要实现的业务的实际场景，包括业务流程、数据表结构、数据表的数据量等。在这里我们需要具体解释我们的业务流程，不要泛泛而谈，最好能够画结构图，把流程解释清楚，同时也要解释我们的数据表设计思路。

（3）测试用例

关于通用测试场景的测试用例，我们还是以数据库调研为例。这里我们可以列举出“只写测试用例”“只读测试用例”“只更新测试用例”三种情况，所以对于业务测试场景，我们需要根据业务的实际场景来操作，例如对读写并发这样的业务进行用例设计，需要开启多个线程或者客户端同时执行业务 SQL 语句。

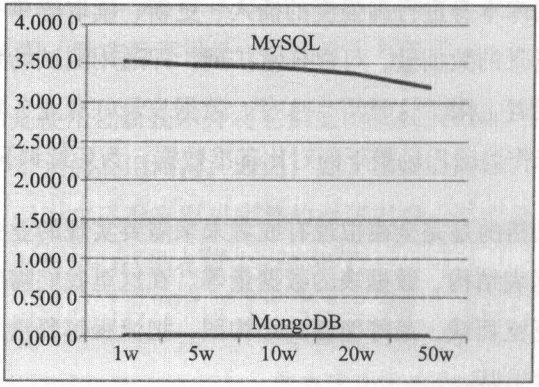
4. 执行方案

由于前一节中设计了测试用例，所以我们这里的执行方案就需要执行测试用例。当然，需要首先完成测试程序的代码编写工作，并完成单元测试。

在通用测试场景下，我们需要按照业务的数据表形式，在各个调研数据库中分别创建这些数据表。注意，这里只是确保数据完整性，并不一定是完全参照我们的业务实际数据表设计方式。针对通用场景执行插入、读取、更新操作，以每秒可以执行的原子化操作（TPS）作为对比值，例如读取 1 万条数据的测试用例执行、插入 100 万条数据的测试用例执行、更新 1 万条数据的测试用例执行。

在业务测试场景下，我们一般需要按照业务流程执行测试用例，例如插入、读取、更新操作是并行执行的，而不是像通用场景下只有单一操作。

无论上述哪一点，执行结束后的输出数据都需要保存下来，最好是采用图表方式进行展示。



5. 讨论结论

调研结论是调研项目的重点部分，我们需要清楚地描述我们的数据对比以及结论。

我们首先需要对技术的设计原理、参数进行对比，以数据库为例，对比内容如下所示：

- 是否开源协议；
- 是否支持分布式；
- 分布式设计架构原理；
- 是否支持事务；
- 数据类型；
- 存储方式；
- 查询方式；
- 稳定性。

接下来，我们需要针对给定场景的对比测试结论进行分析并且描述清楚，列举每一项技术是否适用于我们设计的场景，并给出具体原因。

然后我们需要进行总结，需要覆盖性能角度、技术评测角度、需求方角度、产品化角度以及其他因素等。

- 性能角度：分为通用场景和业务场景，描述哪种技术的性能最佳。以数据库为例，需要说明具体插入、读取、更新，分别哪种技术最强，差距多少，为什么会有这样的差距。
- 技术评测角度：这一栏基本是对于技术的设计原理、参数这一块的综述，需要用简洁的语句描述清楚哪一种技术最适用，为什么；
- 需求方角度：将用户提出的几种业务需求场景转化为技术场景后并进行描述，说明哪种技术最适用，为什么；
- 产品化角度：如果技术不适用于产品，我们需要按照产品优先级来分析、处理问题；

❑ 其他因素：针对一些不属于以上几个角度的因素逐一列举、对比、总结。

以上所有过程全部完成之后，我们需要选择某一项或某两项技术、框架、方案作为下一阶段的工作技术基础。下一阶段可以是预研项目，也可以是产品开发项目，我个人推荐是预研项目，这样的流程较为合理，风险较小。

4.3 技术预研

4.3.1 技术预研思路

在产品规划的指引下，难度较大的关键技术的预研将在项目立项之前以技术预研项目的方式开展。待项目正式立项后，难度较大的关键技术已经攻克，后续产品开发团队的职责是集中更多资源，在非常短的时间内开发高品质产品并推向市场。这就是预研项目的意义。

1. 了解动机

立项预研项目之前，我们需要首先明确以下两点：

1）是否需要立项预研项目：你所困惑的是不是属于预研项目范畴？技术上一定需要预研吗？

2）是否有实际的需求：你目前技术上或者方案上满足不了了，必须预研？

立项预研项目一定是有背景存在的，大多数情况是当前的技术或者方案无法满足用户需求，或者一个全新的产品形态在产品经理脑海中出现，经过头脑风暴发现还存在若干技术或者方案选型难点，如果贸然立项，很有可能出现产品开发延期的情况。因此，需要在产品开发项启动前，明确存在疑惑的技术点。这时候，预研项目就可以启动了。

也有另外一种情况。在技术或者方案调研完毕后，我们列出了下一步计划，一般来说就是预研项目计划，即对调研项目所产生的成果物进行进一步的实践，通过局部实现方式验证调研过程的正确性。

无论哪一种，最终都来源于需求。没有需求，什么都不会启动。

2. 明确目的

首先需要了解用户需求，例如用户提出需要系统具备横向扩展能力，这一点就是很明确的需求。了解需求后，我们需要对现有系统进行剖析，是否支持横向扩展，如果不支持，是哪里存在问题，逐个模块分析后找到最终原因。接下来需要对当前情况进行解释，最好配有设计图或者测试数据，这样可以让读者更加直观了解当前情况。

情况了解清楚之后，我们可以讨论、总结预研目的，例如提出一种新的系统架构方案、采用新的开源框架等，最终的目的是满足用户的需求。因此，不能满足用户需求的预研项目成果是无效的。

3. 确定步骤

预研项目一般来说分为以下几个步骤：

1）搜索预研需求。

2）明确预研目的。

3）确定预研方案，需要针对各个方案（包含当前方案）进行各维度对比，指出每个方案的设计和实现原理，指出存在的不足。然后结合方案进行具体的实现，并测试数据。

4）根据方案的对比、测试数据对比，我们可以明确提出的方案是否适用于用户需求，如果适用，可以推荐进行产品化立项；如果不适用，需要找到备选方案继续预研。如果实在找不出来（这种情况很少出现），需要反思用户需求是否合理，可以继续采取头脑风暴方式。

4.3.2 预研过程

1. 技术介绍

明确预研需求之后，我们需要对所选择的方案或者技术进行介绍，也就是定义和综述。定义指的是对基本概念进行介绍，而综述指的是对当前该技术的发展

及应用进行介绍。综述应注意以下几点：

- 技术研究设定的工作目标应可验证；
- 明确技术适用的范围，以便于后续研发项目的可行性研究；
- 指明研究的局限性，明确写出后续待完成及验证的研发工作；
- 在进行技术研究时，应注意借鉴过程资产库中的经验库，以及可重用和通用的模块库等；
- 当有多种技术路线需要研究、比对时，应在关键技术分析报告中对各技术路线进行对比。

除此之外，我们还需要对该技术应用的参考文献逐一列举，以说明自己是参考了大量论文或文章后才得到预研技术或方案的。

2. 明确方案

这一环节我们需要列举所有预期可能满足需求的解决方案，重点说明需要被本预研过程验证的一个或多个解决方案，并说明其各自的优缺点，以及提供潜在改进方向和应用可能性。

明确方案也需要有大量的论证，为什么你选择了 A 方案，而不是 B 方案？它们两者之间的优缺点是什么？针对业务场景的适用度如何？例如对于分布式系统，我们可以列举单体型架构和微服务架构。两种方案列举后，我们可以将两种方案都实现并对比，也可以采用排除法。如果采用排除法，需要先对比两种方案，例如对比两种架构的优缺点。

采用哪种方案，不能只看技术先进性，还是需要结合自己的实际情况进行预研。注意，只有首先明确方案，才能执行方案。

3. 执行方案

根据前面确定的方案，我们需要进行预研方案的论证，这个过程一般分为若干阶段，采用方案列举—论证—推翻—再列举，或者总体方案—模块方案等方式进行阶段划分。

执行方案的第一步是方案列举，这一步需要做到深入理解。如果你对方案的实现原理不了解，那么无论怎么解释都是很空洞的。第二步是论证，需要从整体开始论证，逐渐下沉到各个重要模块，只有重要模块论证通过，才能进入下一步。第三步是尝试推翻论证的结果，这一步建议召开部门内部评审会，让不同角色发声，你需要逐一解释。如果出现解释模糊的情况，那么说明你的方案存在疑点，需要回炉重做。只有这三步论证都通过，才能进入讨论结论步骤。

4. 讨论结论

预研工作的输出为技术可行性分析报告（技术介绍文档、技术环境搭建文档、功能验证文档、性能测试），有优越性和风险性评估。着重探索和解决技术实现的可行性，使得能够在需要时为产品开发提供支撑。

输出的过程如下：

1）根据公司技术规划和产品研发确定预研技术方向；

2）收集和整理有关预研技术的论文、专利、标准，了解预研技术的现状和未来发展方向；

3）对预研技术原理、技术方案进行全面、深入的分析，关键技术进行仿真和验证；

4）对研究成果进行总结，形成专利、标准提案；

5）编写预研技术可行性报告，为公司产品规划和研发提供决策依据。

需要通过各种数据对比、架构对比、原因剖析等方式，给出被排除的方案的排除原因，对被选择的方案应明确说明其优缺点。

4.4　其他相关讨论

4.4.1　各种开源协议介绍

现今存在的开源协议很多，而经过 Open Source Initiative（OSI）组织批准的

开源协议目前有 58 种（http://www.opensource.org/licenses /alphabetical）。常见的开源协议如 BSD、GPL、LGPL、MIT 等都是 OSI 批准的协议。如果要开源自己的代码，最好也是选择这些被批准的开源协议。

这里我们来看 4 种最常用的开源协议及它们的适用范围，供那些准备开源或者使用开源产品的开发人员 / 厂家参考。

1. BSD 开源协议

BSD 开源协议（Original BSD License、FreeBSD License）是一个给予使用者很大自由的协议。使用者可以“为所欲为”，可以自由地使用、修改源代码，也可以将修改后的代码作为开源或者专有软件再发布。

但“为所欲为”的前提是，当你发布使用了 BSD 协议的代码的产品、或者以 BSD 协议代码为基础对自己的产品做二次开发时，需要满足 3 个条件：

（1）如果再发布的产品中包含源代码，则新的源代码必须带有原来代码中的 BSD 协议。

（2）如果再发布的只是二进制类库 / 软件，则需要在类库 / 软件的文档和版权声明中包含原来代码中的 BSD 协议。

（3）不可以用开源代码的作者 / 机构名字和原来产品的名字做市场推广。

BSD 代码鼓励代码共享，但需要尊重代码作者的著作权。BSD 由于允许使用者修改和重新发布代码，也允许使用或在 BSD 代码上开发商业软件并进行发布和销售，因此是对商业集成很友好的协议。而很多的公司企业在选用开源产品的时候都首选 BSD 协议，因为可以完全控制这些第三方的代码，在必要的时候可以修改或者进行二次开发。

2. Apache Licence 2.0

Apache Licence（Apache License, Version 2.0、Apache License, Version 1.1、Apache License, Version 1.0）是著名的非盈利开源组织 Apache 采用的协议。该协议和 BSD 类似，同样鼓励代码共享和尊重原作者的著作权，同样允许代码修改、

再发布（作为开源或商业软件）。需要满足的条件也和 BSD 类似：

（1）需要给一份 Apache Licence。

（2）如果你修改了代码，需要在被修改的文件中说明。

（3）在延伸的代码中（修改和有源代码衍生的代码中）需要带有原来代码中的协议、商标、专利声明和其他原来作者规定需要包含的说明。

（4）如果再发布的产品中包含一个 Notice 文件，则在 Notice 文件中需要带有 Apache Licence。你可以在 Notice 中增加自己的许可，但不可以更改 Apache Licence 构成。

Apache Licence 也是对商业应用友好的许可。使用者也可以在需要的时候修改代码来满足需要并作为开源或商业产品发布 / 销售。

3. GPL

我们很熟悉的 Linux 就是采用了 GPL（GNU General Public License）。GPL 协议和 BSD、Apache Licence 等鼓励代码重用的许可很不一样。GPL 的出发点是代码的开源 / 免费使用和引用 / 修改 / 衍生代码的开源 / 免费使用，但不允许修改后和衍生的代码作为闭源的商业软件发布和销售。这也就是我们能用免费的各种 Linux 的原因。

GPL 协议的主要内容是只要在一个软件中使用（“使用”指类库引用，修改后的代码或者衍生代码）GPL 协议的产品，则该软件产品也必须采用 GPL 协议，既必须也是开源和免费的。这就是所谓的“传染性”。GPL 协议的产品作为一个单独的产品使用没有任何问题，还可以享受免费的优势。

由于 GPL 严格要求使用了 GPL 类库的软件产品必须使用 GPL 协议，所以商业软件或者对代码有保密要求的部门就不适合集成 / 采用 GPL 协议作为类库和二次开发的基础。

其他细节还有如再发布的时候需要伴随 GPL 协议等和 BSD/Apache 等类似。

4. LGPL

LGPL（GNU Lesser General Public License）是 GPL 中一个主要被类库所使用的开源协议。它和 GPL 要求任何使用、修改、衍生自 GPL 类库的软件必须采用 GPL 协议有所不同，LGPL 允许商业软件通过类库引用（link）方式使用 LGPL 类库而不需要开源商业软件的代码。这使得采用 LGPL 协议的开源代码可以被商业软件作为类库引用并发布和销售。

但是如果修改 LGPL 协议的代码或者衍生代码，则所有修改的代码中涉及修改部分的额外代码和衍生的代码都必须采用 LGPL 协议。因此 LGPL 协议的开源代码很适合作为第三方类库被商业软件引用，但不适合希望以 LGPL 协议代码为基础，通过修改和衍生的方式做二次开发的商业软件采用。

GPL/LGPL 都保障原作者的知识产权，避免有人利用开源代码复制并开发类似的产品。

4.4.2 对技术发展的预估

作为一名技术团队的管理者，你必须了解当前技术发展趋势，并能够对趋势进行分析，进而明确自己团队未来 1 年、3 年，甚至是 5 年、10 年的发展目标。要拥有这一能力很难，但是你不得不做到。

我们可以通过科技网下载所需要的论文，这一点可以帮助我们了解当前国内较新的研究成果。而对于国外论文，我们可以通过 Google Scholar 进行检索，也可以通过一些国外比较流行的科技网站来进行信息获取、知识学习，这些都有助于我们建立自己的知识储备体系，帮助我们完成对技术发展的预估。

除了搜索论文以外，我们也可以通过关注一些大咖或技术企业的公众号，了解最新的科技方向。还有就是技术峰会，这类需要自己筛选，有一些会议是以宣传自身产品为目的的，并不是为了做技术分享，这些需要自己体会。

4.4.3 开源的好处

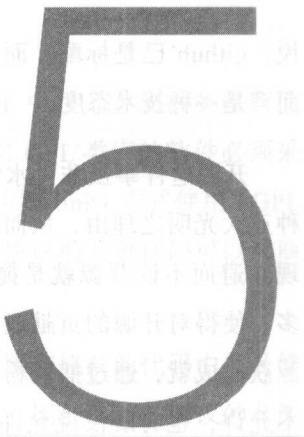
在开发的圈子里，开源已成势，无论公司大小都在开源。个人开发者更不必

说，github 已是标配。而开源与使用 NodeJS 一样，对个人而言是潮流，对团队而言是一种技术态度。

开源这件事虽非洪水猛兽，但因为开源可能会为一些人转移公司代码提供一种正大光明之理由，从而对公司造成损失，所以在公司层面上很难说清利弊。但现在避而不谈开源就是掩耳盗铃。开发者及各公司使用开源软件的情况越来越多，使得对开源的贡献成为相当一部分技术人员的一种技术理想。他们希望从开源获得成就，通过捕获粉丝以获得崇拜感。所以，有的公司允许开源，有的公司不允许，也有既没说允许也没说不允许的。一般允许开源都会有相应审核机制，对开源的选择权主要取决于每个部门自己的考虑。比如现在的 BAT，开源都有相应的审核机制。

开源不仅意味着公司技术有了影响力，而且开源后技术需求的输入会变多，外部会给内部提供许多技术需求，从而通过从外部推进内部加快技术产出与技术创新。创新后再回归到开源，进而构成技术闭环。需求持续输入可让技术像产品一样迭代升级，提升功能单一的技术生命周期；需求多样化可以提高创新能力，从而让技术更有生命力。

在获得影响力之后，简历的收集渠道会扩宽，会有同行主动给你发简历；同时也会给现在团队成员带来平台的成就感，也能让外界技术人员对团队成员产生认同感，这对于技术人员来说非常重要。



第 5 章 系统架构

软件技术学习到一定的程度后，会面对软件架构师这一门槛。一直以来，在软件行业内部，对于什么是架构有很多的争论，每个人都有自己的理解，甚至很多架构师一说架构，就开始谈论应用架构、硬件架构、数据架构等。而事实上，架构在软件发明前就早已存在了。由于众说纷纭，莫衷一是，于是大家产生了很多困惑。软件并不是虚无缥缈的东西，它和现实生活是紧密相关的。业务和架构都是处理人的问题，只有和人的问题联系在一起，才是真正的系统架构设计。

本章主要介绍和解决以下问题：

- 什么是系统架构、软件架构师，他们的工作内容、责任、思路是什么。
- 系统架构案例介绍、常用工具介绍。
- 系统架构能力培养。
- 架构过程常见问题分析。
- 其他与系统架构相关知识分享。

5.1 系统架构工作

5.1.1 软件架构概念

1. 什么是软件架构

谈到软件架构，很多人会认为软件架构就是一堆框架的组合，其实不对，软件架构本身是对于软件实体组织形式的阐述，使用框架的意义是快速完成软件架构设计，而不是取代软件架构设计，两者本质上是不一样的，它们的关系更像是设计图纸和所使用的原材料的关系。软件架构就是通过对软件生命周期的拆分，在符合业务架构的前提下，达到软件本身访问增长目的的方式。这个增长需要软件开发的增长，也需要软件运行的增长，进而支撑业务的增长。

软件架构离不开软件开发团队的组织架构，这个组织架构是软件开发生命周期和软件运行生命周期的执行者。离开了组织架构，任何软件架构设计都是纸上谈兵，因为架构的核心生命周期就是架构的执行。

2. 工作分工

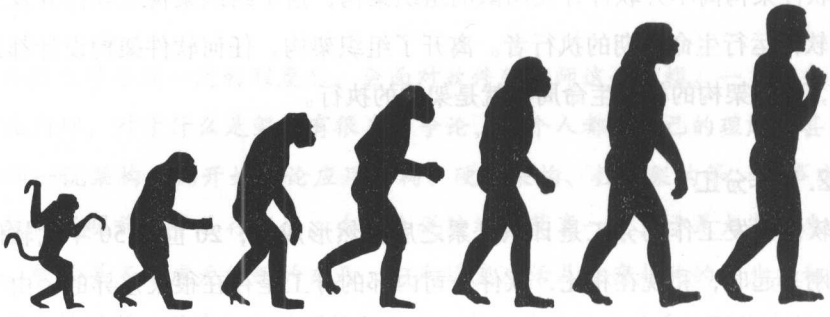
软件开发工作的分工是日积月累之后自然形成的，20 世纪 50 年代软件开发工作刚兴起时，和现在相比，软件公司内部的分工是存在很大差异的。由于各种外部因素及其他约束，造成软件开发需要分工，自然而然就会分出不同的角色做不同阶段的事情，这就是需求决定最终概念。分工之后，有些人收集并汇总需求，有些人编写代码，有些人执行测试等。这样分工比较容易，招人也容易，切分出来的工作门槛也会比较低。分工清晰后，会给创业公司带来一定的麻烦，因为招聘岗位太多，造成薪资负担很重，希望能够招到牛人，什么都能一人干完。所以现在又开始流行全栈工程师，这类人能够胜任从产品到技术到测试所有工作，其实这是自然架构的一种倒退，必然会遇到问题。

3. 什么是组织架构

前面提到了组织架构。软件架构若把软件看成虚拟的人，那软件架构实际上就是虚拟人的组织架构。架构拆分的原则，首先来源于业务自身的组织架构，软

件架构要保持和业务组织架构的匹配关系。其实也只有软件架构和组织架构保持匹配，软件架构设计才能真正落地，软件架构师才能真正拥有工作的成就感、归属感。其次来源于软件开发团队自身的组织架构，没有组织架构的软件团队是不存在的，即便是现在流行的扁平化组织架构管理体系，组织架构依然是存在的，只是层级被压缩到较少程度而已。最后来源于用户的流量对软件本身的冲击，这其实是软件模拟人类行为的一种方式，一个人使用和成千上万人同时使用的业务场景都属于正常情况。如果软件开发团队的组织架构和业务的组织架构能一直保持高度一致，内部损耗就会达到最小，软件的架构会更简单，软件架构有效落地的可能性也会大幅度增加，软件模拟人的行为也会更加真实。

4. 进化



很多人认为软件架构是进化或演化出来的。进化的含义是由一个物种变成另外一个物种，是本质的变化，比如物种由水生进化到了陆生等。

回到技术，架构追求的实际上是业务不断长大：通过对业务生命周期的拆分，突出并精简业务核心生命周期，弱化非核心生命周期，达到不同生命周期可以在空间和时间上并行，便于不同的人同时开展工作，提升业务人员单位时间内的产出。比如说我们设计的分布式存储系统，理论上通过增加机器就能无限提升存储能力和并发能力，也就节省了核心生命周期的开发时间，提升了开发人员的开发效率。

业务相当于基因，而架构呈树状延展则相当于细胞的分裂。就好比每一个

人，都是从一个细胞逐渐分裂出来的。一个细胞最终分裂成什么，起决定作用的不是分裂本身，而是它的基因。基因决定了细胞最终会分裂生长成什么样的一个生命。比如一棵树从种子长成小树苗，再长成参天大树，这不叫进化，这叫生长。长成什么树是由树的基因决定的，不是架构。因为不管怎么拆分，业务的目标，也就是基因没有任何变化。如果一个企业的业务由制造商变成电商，这个时候业务就发生进化了。因为基因变了，业务已经完全不一样了，整棵架构树的含义就变了。该企业的软件架构就需要重新设计，而不能在原有的架构上修改，也就不存在架构的进化。

严格来说，只有业务才会进化，架构是支撑业务长大的。业务的核心生命周期相当于架构树的主干，主干没有变化，则说明没有变成一棵不同品种的树，因此只有架构的拆分不能叫作进化。随着架构拆分得越来越细，“树”长得越来越大，并行度越来越高，业务也就长得越来越大。

5.1.2 软件架构师岗位

1. 为什么需要架构师?

我们需要有人能够回答这些问题：

- ❑ 我们这个系统的边界是什么?
- ❑ 我们系统由哪几部分组成?
- ❑ 各模块之间怎么通信?
- ❑ 选择什么样的基础技术?
- ❑ 为什么要这样选择?
- ❑ 技术方案未来会遭遇哪些坑?
- ❑ 我们的备选方案有哪些?
- ❑ 从技术角度这个应用将来如何持续扩展功能?

这一系列的问题归咎为规划，因为做所有事情都需要规划，对于研发流程而言，规划阶段就是我们的总体设计阶段。一个复杂的软件系统更需要规划，如果

没有规划可能连一行代码都很难写出来（当然会有人持反对意见，因为他们用一堆框架搭起一个系统，虽然没有设计环节，但是貌似还能正常运行。但是别高兴得太早，当你遇到业务急速扩充，系统出现各种异常情况时，你会被拖垮）。这个问题从一个程序员的角度去思考，其实也是一样的，你在写一个上万行的代码时难道是想到哪里就写到哪里吗？

2. 职责

架构师要解决的问题都是人的问题。更进一步，架构师要解决的问题基本都是别人的问题。别人的问题解决了，架构师自己的问题才能解决。任何找上架构师的问题，基本都不是真正的问题。为什么呢？因为如果是真正的问题，提问题的人肯定能够自己解决，不需要找架构师。因此，架构师都要有觉悟，理解并发现问题永远比解决问题更加重要，遇到问题首先进行分析，不要急于解决问题。

3. 谁是架构师？

很多公司设了软件架构师的职位，主要职责是做出架构设计，他们具备一定的影响力，但并不具备调动组织架构的权利。这样的职位往往不能发挥架构师的作用，有时候还会起反作用。软件架构师必须是一个组织的领导人，有权利调整这个组织的架构，才能够更好地发挥架构师的作用，才能够把软件开发生命周期、软件运行生命周期和业务生命周期的拆分落实执行。软件开发团队的组织领导人其实都是架构师，只是没有这个头衔而已，真正的架构师不一定需要具备架构师的头衔。

一个开发团队的领导，如果不能在架构方面给出意见，那就不能在整个设计环节给出自己的观点或者对团队进行指导，也就可能会在整个软件开发流程中脱节，进而丢失自己的技术领导力，也就做不成技术管理者。一名优秀的技术管理者，技术在前，管理在后，并不是说两者有太大的轻重差异，而是你需要花费 70% 的时间在技术上，只能花 30% 的时间在管理上，但是你需要用这 30% 的时间做完 100% 的管理工作。技术、管理，一样都不能差，架构能力也是一样。因此，一个好的领导就是一个很好的架构师。我相信，在架构师的领导下，这个组

织一定是健康向上的。

当然，也可以有另一种方式来允许架构师角色独立存在。架构师可以由专门的个人或者团队组成，他们承担新技术、框架的调研工作，负责对用户提出的需求进行评估，采用新的技术做出产品原型、输出技术调研报告，供产品部门在技术选型和技术架构选型时参考，这也可以体现架构师的水平和贡献。

5.1.3 软件架构师的责任

一个软件往往因为某个业务需求而产生，后续不断更新、修改，逐渐变异、长大，当该软件不再被需要（因为业务的消失）或有更好的软件来替代时就会被废弃，完成使命而消亡。软件的整个生命周期也会发生切分，从而形成两个子生命周期，即软件开发生命周期和软件运行生命周期。

作为软件架构师，必须时刻把对软件生命周期和业务生命周期的识别放在第一位。软件生命周期的核心在于软件运行生命周期，以及围绕软件运行生命周期的拆分和组织，业务生命周期的核心在于围绕业务核心生命周期的拆分和组织。

对于软件生命周期，必须要深入思考软件开发生命周期和软件运行生命周期。在这个基础上，要根据业务的情况合理地进行软件开发生命周期的架构拆分和软件运行生命周期的架构拆分。软件开发的拆分和软件运行的拆分的目标都是支持业务流量的增长。

在代码层面，要对业务代码和访问代码做好架构拆分，要确保业务代码符合业务的生命周期，使得业务生命周期活动的结果积累在生命周期的主体上，也就是要具有内聚性，避免散落到访问代码中。这样软件的拆分就不会有太大的问题。

架构在软件发明前就已经存在很久了，我们应该多向大自然、艺术界、建筑界学习架构的概念，因为软件并不是虚无缥缈的事物，它和我们的现实生活是紧密相关的，它来源于生活，最终又通过软件服务回到生活。企业软件架构的设计

不仅要注重某一个系统功能，更需要给企业一个可进化的、可持续发展的、不断创新的平台。

团队达到一定规模之后，技术管理者（也可能有独立架构师存在）的大部公时间就需要花费在思考上面了，当然也可以继续编程，但是编程的目的是验证架构是否合理，所以不要受是否需要编程这一思维的束缚。如果设计得不好，那么团队就会走很多弯路，如果想要设计得很好，你必须自己或者带领团队做很多的测试、预研工作。作为架构师，你需要多多思考，很多时候因为很忙、事情很多，导致真正思考时间太少了。

5.1.4 软件架构思路

1. 前提条件

软件架构师需要学会拆分生命周期，并且应该指导形成组织架构来落实架构的执行，需要平衡别人的利益，甚至去调整别人的利益。对于软件架构师而言，是无法直接去调整业务利益的，只能够在某种程度上去影响。毕竟软件架构师是为业务服务的，而不是主导业务的业务架构师。

2. 应对增长

架构的目的就是为了增长，而要达到增长，就必须让很多人合作为完成同一件事情，并且使他们做的事情合并起来达到 1+1>2 的效果，最少也要达到 1+1=2 的效果。

架构师应该把需要增长的业务了解清楚，挖掘出核心生命周期，并确定核心生命周期的主体。换句话说架构师要发现问题的主体，并确定核心问题。在确定业务核心生命周期以及核心生命周期的主体之后，架构师还需要对业务核心生命周期进行分析，剥离出非核心生命周期，并根据当前人员的状况，合理地分配非核心生命周期的权责。这样不同的人就可以并行且互不影响地做不同的事情，最后根据核心生命周期，把他们的工作成果组合起来，达到 1+1>2 的效果。

架构师工作的反馈应该由问题的解决效果，也就是增长的效果来决定。如果

以很少的资源达到了很大的业务增长，那肯定是一位好的架构师，公司所节省下来的资源应该回馈给架构师，从而形成正向反馈。从这一点来看，架构师要对总体增长的效果负责。

5.1.5 软件架构落地

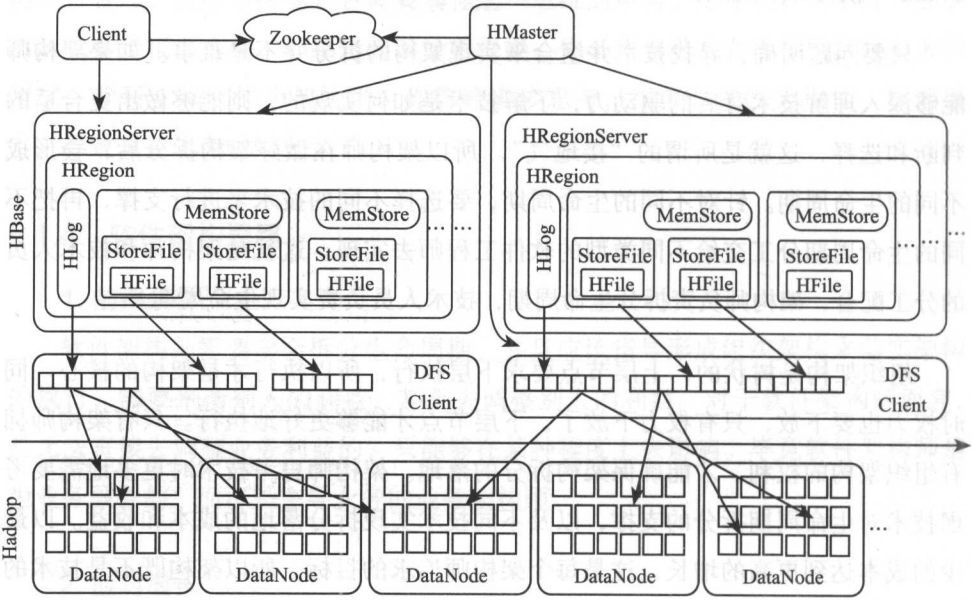
只要问题明确，寻找技术并组合来实现架构的拆分并不是难事。如果架构师能够深入理解技术背后的驱动力，了解技术是如何实现的，则能够做出更合适的判断和选择，这就是所谓的“接地气”。所以架构师在做好架构拆分后，会形成不同的生命周期。针对不同的生命周期，要选择不同的技术来进行支撑，再把不同的生命周期分工交给不同类型的软件工程师去实现。这就是架构师和技术人员的分工配合，架构师负责拆分生命周期，技术人员负责实现生命周期。

组织架构是树状的，上层节点要求下层执行，所以执行才是架构的核心。同时权力也要下放，只有权力下放了，下层节点才能够更好地执行。只有架构师拥有组织架构的权利，才能确保架构拆分的落地。架构师思考技术时更多地需要考虑技术对生命周期拆分的支撑，以及不同技术实现拆分落地的成本和收益。以最少的成本达到更高的增长，这是每个架构师追求的目标，所以架构既不是技术的对立面，也不是技术的同一面，两者是相辅相成的关系。

很多软件架构设计都有相通性。我们来看看 HBase 的 RegionServer 设计方式。在 HBase 内部，所有的用户数据以及元数据的请求经过 Region 的定位后，最终会落在 RegionServer 上，并由 RegionServer 实现数据的读写操作。RegionServer 是 HBase 集群运行在每个工作节点上的服务。它是整个 HBase 系统的关键所在，一方面它维护了 Region 的状态，提供了对于 Region 的管理和服务；另一方面，它与 Master 交互，上传 Region 的负载信息，参与 Master 的分布式协调管理。HBase 系统架构如下页图所示。

HRegionServer 与 HMaster 以及 Client 之间采用 RPC 协议进行通信。HRegionServer 向 HMaster 定期汇报节点的负载状况，包括 RS 内存使用状态、在线状态的

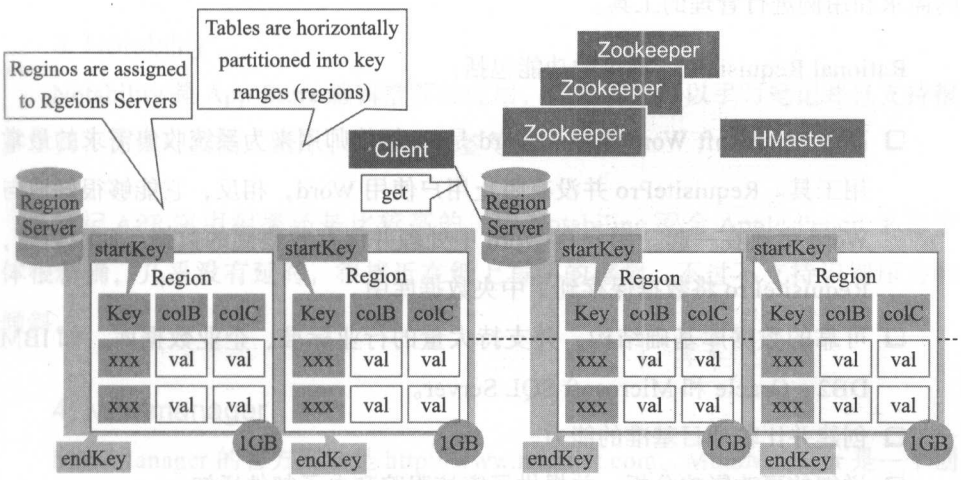
Region 等信息，在该过程中 HRegionServer 扮演了 RPC 客户端的角色，而 HMaster 扮演了 RPC 服务器端的角色。HRegionServer 内置的 RpcServer 实现了数据更新、读取、删除的操作，以及 Region 涉及的 Flush、Compaction、Open、Close、Load 文件等功能性操作。



Region 是 HBase 数据存储和管理的基本单位。HBase 使用 RowKey 将表水平切割成多个 HRegion，从 HMaster 的角度，每个 HRegion 都记录了它的 StartKey 和 EndKey（第一个 HRegion 的 StartKey 为空，最后一个 HRegion 的 EndKey 为空），由于 RowKey 是排序的，因而 Client 可以通过 HMaster 快速定位每个 RowKey 在哪个 HRegion 中。HRegion 由 HMaster 分配到相应的 HRegionServer 中，然后由 HRegionServer 负责 HRegion 的启动和管理，和 Client 的通信，负责数据的读（使用 HDFS）。每个 HRegionServer 可以同时管理 1000 个左右的 HRegion。

再来看看软件系统架构方面的分区设计。以任务调度为例，假设我们有一个中心调度服务，那么当数据量不断增多时，这个中心调度服务一定会遇到性能瓶

颈，因为所有的请求都会最终指向它。为了解决这个性能瓶颈，我们可以将任务调度拆分为多个服务，多个服务都可以处理任务调度工作。那么问题来了，每个任务调度服务处理的源数据是否需要完全一致？根据华为公司发布的专利发明显示，他们对于每一个任务调度服务由数据来源区分操作，即按照任务调度数量对源数据进行划分，比如3个任务调度服务，源数据按照行号对3取余的方式进行划分，如果运行了一段时间之后，任务调度服务出现了数量上的增减，那么这个取余划分需要重新进行，要按照那个时候的任务调度数量重新划分区间。



上面举的两个示例的实现原理是完全一致的，架构设计也是类似的，只是最终解决的业务问题有所不同。

5.1.6 有用的工具

在软件开发流程的各阶段中，IT架构师都必须不断地评估自己的设计，以确保它能够满足需求。如果需求发生了更改，这在现实世界中是非常普遍的，最终产品的计划和时间安排都必须随之进行相应的更新。使用一些优秀的软件工具，可以使这项工作变得更加容易。让我们来看看在这个周期的各个阶段可以使用的一些工具。

1. IBM Rational RequisitePro

需求分析是系统设计过程中的一个重要部分，系统中所有的利益相关者都通过这项工作来确定客户及软件应用程序自身的要求和需求，这是一个漫长的过程，并且会影响最终产品，所以IT架构师必须确保不遗漏任何一个方面。需求分析和管理工具必须具备完成所有这些任务所需的功能。

IBM Rational RequisitePro解决方案是针对项目组（想要改进项目目标的交流、增强协作开发、减少项目风险，以及在部署之前提高应用程序质量的项目组）的需求和用例进行管理的工具。

Rational RequisitePro提供的功能包括：

- 高级Microsoft Word集成。Word是IT架构师用来为系统收集需求的最常用工具。RequisitePro并没有阻止用户使用Word，相反，它能够很好地与Word集成在一起。用户在Word中处理并保存收集到的数据，而在后台，RequisitePro将数据保存到了中央数据库中。
- 可靠的数据库基础结构，并支持大量的行业标准、企业数据库，如IBM DB2、Oracle和Microsoft SQL Server。
- 创建并比较项目基准的能力。
- 详细的更改影响分析，并提供了审核跟踪和电子邮件通知。
- 深入的可跟踪性和覆盖率分析。
- 可靠的、灵活的、内置的报告系统。
- 为分布式团队提供Web访问的模块。
- 用户定义需求类型的能力。
- 大量可配置的项目和文档模板。

2. ArgoUML

系统设计和体系结构设计工具可以在很大程度上帮助IT架构师完成可靠的和高质量的软件。ArgoUML是一种用于系统设计和架构设计的、优秀的开放源代码工具。它提供了一些有趣的特性，这些特性使得它成为市场中最流行的UML

工具之一。

ArgoUML 具有以下特征：

- 基于开放标准（XMI、SVG 和 PGML）。
- 完全用 Java 语言编写，所以它是平台独立的。
- 开放源代码，这使得它易于扩展，甚至可以根据具体需求对它进行自定义。
- 基于 UML 1.4。

3. Notability

Notability 是 App Store 编辑推荐的应用，作用就是可以手写笔记并且支持很多工具，比如可以插入图片、音频、便签等。

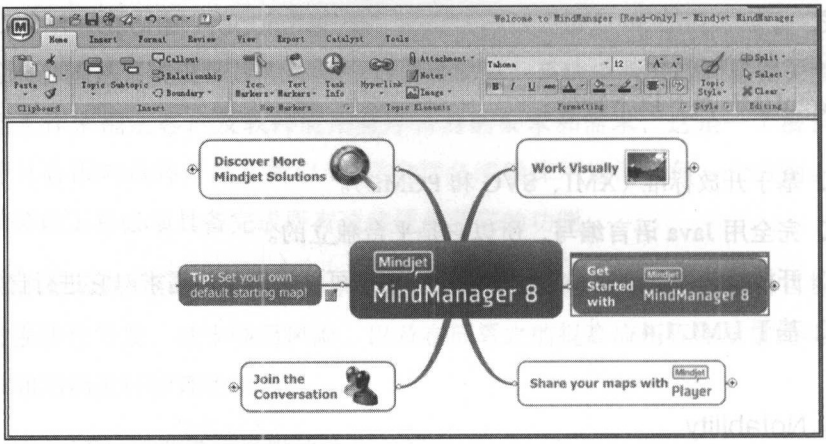
笔记 APP 使用频率还是比较高的，用 Notabiling 配合 Apple Pencil 手写字体很精确，几乎没有延时，很接近在纸上写字的感觉，不过不支持识别压力和倾斜。

4. Mindmanager

MindManager 的官方网站是 http://www.mindjet.com。MindManager 是一个创造、管理和交流思想的通用标准，其可视化的绘图软件有着直观、友好的用户界面和丰富的功能，这将帮助用户有序地组织思维、资源和项目进程。

MindManager 也是一个易于使用的项目管理软件，能很好地提高项目组的工作效率和小组成员之间的协作性。它作为一个组织资源和管理项目的工具，可从脑图的核心分支派生出各种关联的想法和信息。

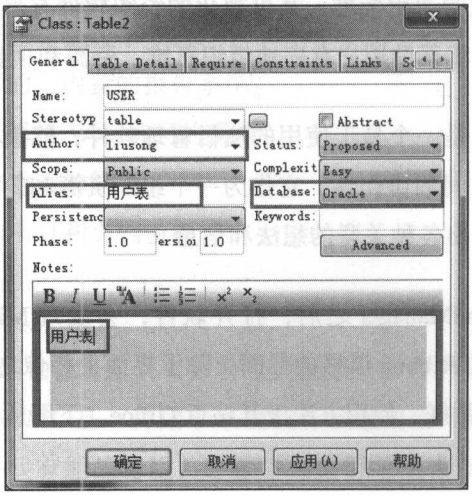
正确安装好 MindManager 之后，打开软件，会发现 MindManager 使用的是和 Office 2007 同样的 Ribbon 风格的界面。除了界面上相像之外，MindManager 和 Office 的互操作性非常好，使用方法极其贴近 Office。下图所示是 MindManager 的界面。



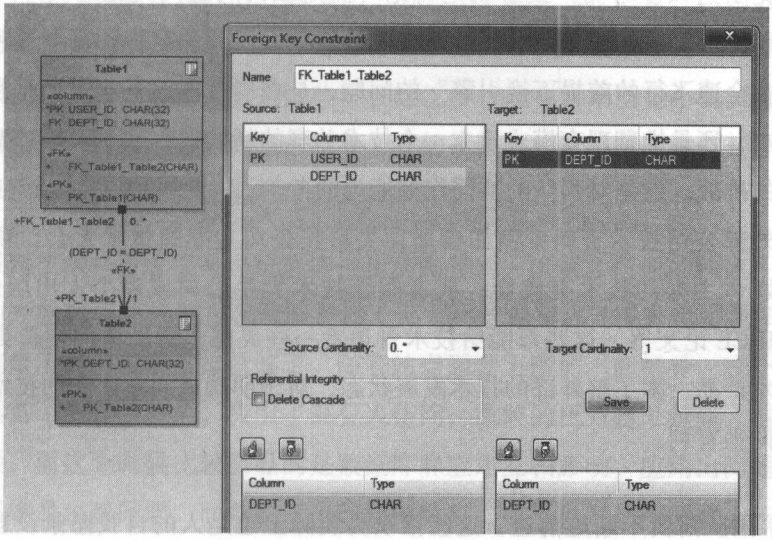
5. Enterprise Architect

Enterprise Architect（简称 EA）和 Rose 是软件开发过程中常用来进行 UML 建模的工具。EA 使用的比较广泛，能够绘制用例图、类图、数据库模型、时序图、协作图、状态图等常用图形。

以一个 user 表为例，下图框中的各项均是要完善的。设置 Alias（别名）后，我们看到的就是中文。Database 的选择涉及后面新建字段后的数据类型以及和数据库的交互。填写 Notes 后，数据库中也会出现注释。Author 标识设计对象。



设计外键关联如下图所示。



5.2 系统架构能力培养

5.2.1 提升驱动力

一个技术团队必须有自己的技术定位，也可以说是有技术愿景，只有这样才能留住对技术很有热情的工程师。明确了技术定位后，我们就需要制定团队技术能力的提升计划，特别是在系统架构能力的提升方面。而提升这些能力，又都需要有实际推动力。一般来说，推动力可能来自以下三个方面：

- 客户实际需求：例如客户需要你设计一个能够支撑 1 万路并发的系统，而通过测试发现现在的设计只能支持 1000 路，那当然需要技术上改进，可能是局部改进，也可能是重构。王坚博士在多个场合都曾经讲过，“双 11 是过去几年阿里技术发展的强大驱动力。”这就说明了业务需求是技术推动力。技术真正发挥价值的地方，一定是对用户产生影响的时候。如果做不到这点，一旦遇到人力紧张的情况，这样的技术改造就会被放弃。

- 测试数据：例如进行性能测试时发现一个数据很可疑，为什么操作数据库花了这么多时间？接着就是讨论可能是编码问题，可能是设计的算法问题，也可能是业务逻辑设计造成的。业务发展到一定阶段都会遇到“飞机在全速飞行的前提下换引擎”的问题，是在现有框架下对两个业务分别改造，还是推翻现有模式建立一个技术共享的新模式？这不仅是对架构能力的挑战，更是对团队的协同作战能力的考验。
- 外部公司技术资料：例如参加顶尖科技公司主办的技术大会、顶尖科技媒体主办的技术交流会，或者阅读技术大牛公司或者个人出版的图书、IEEE 论文等，这些都能给技术管理者一些灵感，帮助他们创造更好的技术产品。多了解外部的技术发展状态，对于明确或纠正团队的技术发展方向很有利。

技术团队需要不断地前进，这些推动力实际上也是人的自我愿景设置，只有不断给自己设置愿景的人，才会不断地尝试提高自己的技术能力、设计能力。

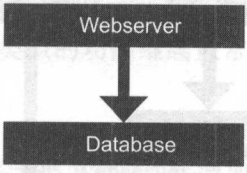
5.2.2 步步为营

“不应该一开始就承担大项目。应该先从小的、无关紧要的项目入手，并且不要去希望该项目能够做大。否则，你就会做出过度的设计，并且通常会把项目的重要性看得比它实际的要高。或者更糟的是，你会被预想的不切实际的项目规模吓得望而却步。因此，要从小规模的项目做起，把细节考虑周全，不要盲目憧憬过大的愿景或虚幻的设计。如果项目不能解决一些相当紧急的需求，那么它十有八九是被过度设计了”。这是林纳斯·本纳第克特·托瓦兹（Linus Benedict Torvalds）的观点。他是著名的电脑程序员、黑客，Linux 内核的发明人及该计划的合作者。

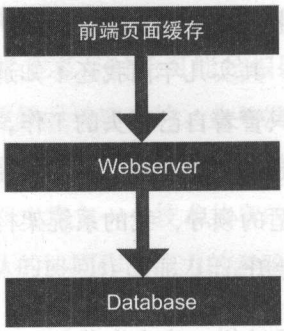
我们以一个架构的实际变化为例来进行说明。一个电商网站的发展过程反映了网站十几年的发展历程，当然也可以理解为一步步构建起一个网站系统架构的过程。虽然我们希望网站一开始就能有一个最好的架构，但事物是在发展中不断

前进的，网站架构也会随着业务的扩大、用户的需求不断完善，在这个过程中设计能力也是一步步在提升的。其实几年前我还不知道真正的设计，因为之前的工作经历分工太明确，每个人只管着自己手头的工作，大型系统的架构设计工作多由别人把控，我们这些人只是做些实现的工作，现在看来就是小工。直到加入了现在的公司，遇到了一位合适的领导，我的系统架构思维瞬间被打开，能力也伴随着思维的不断拓展而快速上升。

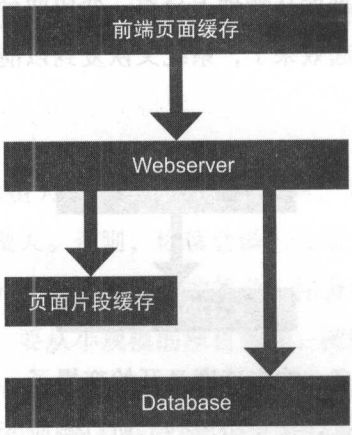
回到技术，系统架构也是这样一步步完善的，以刚才所说的构建一个网站为例，刚开始时可能只是在托管服务器上部署自己的程序（很有可能是单机程序），因为流量不高，能够支撑就够了。随着站点运营力度的不断提升，这个时候由于网站具备了一定的特色，吸引了部分人访问，系统的压力越来越高，响应速度越来越慢，而这个时候比较明显的是数据库和应用互相影响，应用出问题了，数据库也很容易出现问题，而数据库出问题的时候，应用也容易出问题。于是进入了第一个阶段，将应用和数据库从物理上分离，变成两台机器，这个时候技术上没有什么新的要求，但确实起效果了，系统又恢复到以前的响应速度，并且可以支撑住更高的流量。



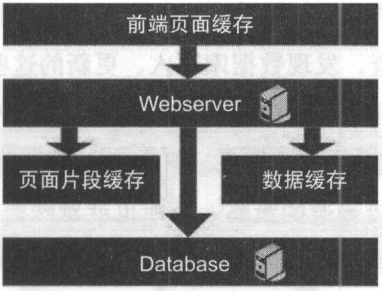
随着访问的人越来越多，响应速度又开始变慢了，查找原因，发现是访问数据库的操作太多，导致数据连接竞争激烈，所以响应变慢，于是进入了第二个阶段。数据库连接又不能开太多，否则数据库机器压力会很高，因此考虑采用缓存机制来减少数据库连接资源的竞争和对数据库读取的压力。这个时候也许首先会选择采用 squid 等类似的机制来将系统中相对静态的页面（例如一两天才会有更新的页面）进行缓存（当然，也可以采用将页面静态化的方案），这样程序上可以不做修改，就能够很好地减少对 WebServer 的压力以及减少数据库连接资源的竞争。



增加了 squid 实现缓存后，系统整体速度确实是提升了，WebServer 的压力也开始下降了，但是随着访问量的增加，系统又开始变慢了，于是进入第三个阶段。在尝到了 squid 之类的动态缓存带来的好处后，我们开始想能不能将现在那些动态页面里相对静态的部分也缓存起来呢？因此考虑采用类似 ESI 之类的页面片段缓存策略，于是开始采用 ESI 来做动态页面中相对静态的片段部分的缓存。



采用 ESI 之类的技术再次提高了系统的缓存效果后，系统的压力确实进一步降低了，但同样，随着访问量的增加，系统又开始变慢，于是进入第四个阶段。经过调查，可能会发现系统中存在一些重复获取数据信息的地方，就像获取用户信息等，这个时候开始考虑是不是可以将这些数据信息也缓存起来，于是将这些数据缓存到本地内存，改变完毕后，完全符合预期，系统的响应速度又恢复了，数据库的压力也再度降低了不少。



好景不长，随着系统访问量的再度增加，Webserver 机器的压力在高峰期会陡增，这个时候开始考虑增加一台 Webserver，进入第五个阶段，这也是为了同时解决可用性的问题，避免单台的 Webserver 宕机没法使用的情况。在做了这些考虑后，决定增加一台 Webserver，但同时会碰到一些问题：

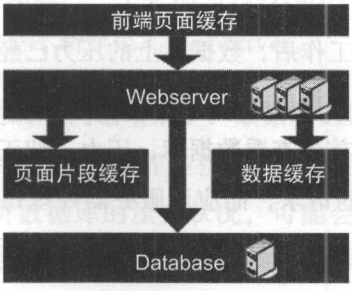
1）如何将访问分配到这两台机器上？这个时候通常会考虑的方案是 Apache 自带的负载均衡方案，或 LVS 这类的软件负载均衡方案。

2）如何保持状态信息的同步，例如用户 session 等？这个时候会考虑的方案有写入数据库、写入存储、cookie 或同步 session 信息等机制。

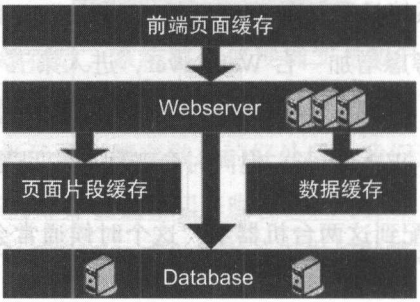
3）如何保持数据缓存信息的同步，例如之前缓存的用户数据等？这个时候通常会考虑的机制有缓存同步或分布式缓存。

4）如何让上传文件这些类似的功能继续正常？这个时候通常会考虑的机制是使用共享文件系统或存储等。

在解决了这些问题后，终于把 Webserver 增加到两台，系统终于又恢复到了以往的速度。



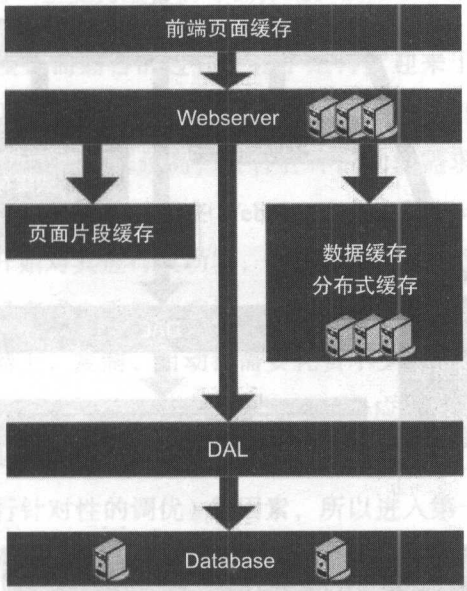
享受了一段时间的系统访问快速响应的幸福后，系统又开始变慢了，这次又是什么状况呢？经过调查，发现数据库写入、更新的这些操作的部分数据库连接的资源竞争非常激烈，导致了系统变慢。这下怎么办呢？进入第六个阶段，此时可选的方案有数据库集群和分库策略，但集群方面有些数据库支持的并不是很好，因此分库是采用比较普遍的策略，分库也就意味着要对原有程序进行修改。修改实现分库后，目标达到了，系统速度甚至比以前还快了。



随着系统的不断运行，数据量开始大幅度增长，这个时候发现分库后查询仍然会有些慢，于是按照分库的思想开始做分表的工作，进入第七个阶段。于是萌生出能否增加一个通用的框架来实现分库分表的数据访问的想法，此时不可避免地会需要对程序进行一些修改，也许在这个时候就会发现自己要关心分库分表的规则等，这个演变的过程相对而言需要花费较长的时间。当然，也有可能这个通用的框架会等到分表做完后才开始做。同时，在这个阶段可能会发现之前的缓存同步方案出现问题，因为数据量太大，导致现在不太可能将缓存存在本地再进行同步，故需要采用分布式缓存方案。于是，又是一通考察和折磨，终于将大量的数据缓存转移到分布式缓存上了。

在做完分库分表这些工作后，数据库上的压力已经降到一个比较低的点，又开始过着每天看着访问量暴增的幸福生活了。突然有一天，系统的访问又开始有变慢的趋势了，这个时候首先查看数据库，压力一切正常，之后查看 Webserver，发现 Apache 阻塞了很多的请求，而应用服务器处理每个请求也是比较快的，看来是请求数太高需要排队等待，导致响应速度变慢，于是进入第八个阶段。添加

一些 Webserver 服务器，在添加 Webserver 服务器的过程中，有可能会出现几种挑战：

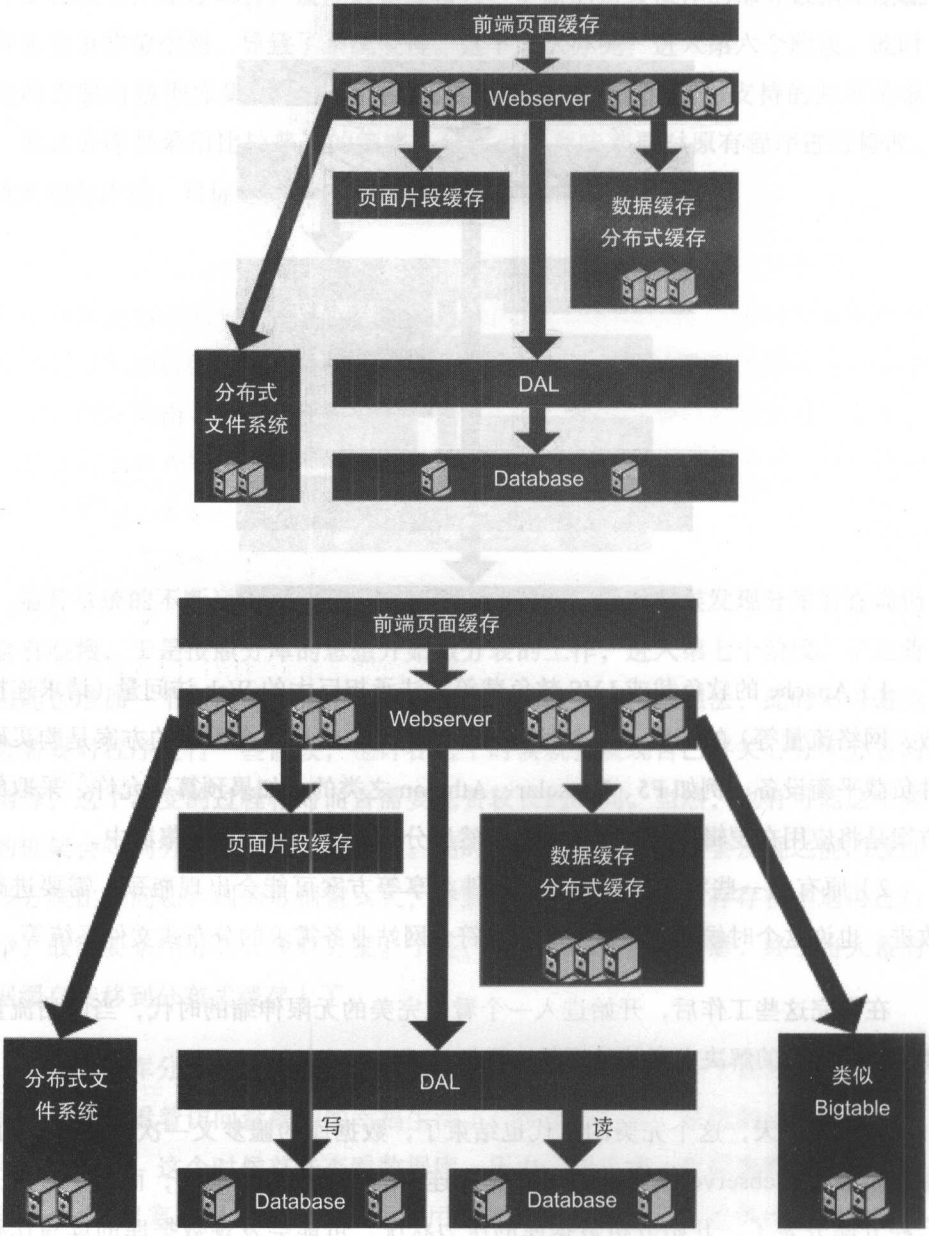


1）Apache 的软负载或 LVS 软负载等无法承担巨大的 Web 访问量（请求连接数、网络流量等）的调度，这个时候如果经费允许的话，会采取的方案是购买硬件负载平衡设备，例如 F5、Netsclar、Athelon 之类的。如果预算不允许，采取的方案是将应用在逻辑上做一定的分类，然后分散到不同的软负载集群中。

2）原有的一些状态信息同步、文件共享等方案可能会出现瓶颈，需要进行改进，也许这个时候需要根据情况编写符合网站业务需求的分布式文件系统等。

在做完这些工作后，开始进入一个看似完美的无限伸缩的时代，当网站流量增加时，应对的解决方案就是不断地添加 Webserver。

突然有一天，这个完美的时代也结束了，数据库的噩梦又一次出现在眼前。由于添加的 Webserver 太多，导致数据库连接的资源还是不够用，而这个时候又已经分库分表了，开始分析数据库的压力状况，可能会发现数据库的读写比很高，这个时候通常会想到采用数据读写分离的方案。当然，这个方案要实现并不


前端页面缓存
Webserver
页面片段缓存
数据缓存
分布式缓存
分布式
文件系统
DAL
Database
前端页面缓存
Webserver
页面片段缓存
数据缓存
分布式缓存
分布式文
件系统
DAL
写
读
类似
Bigtable
Database
Database

容易，另外，可能会发现一些数据存储在数据库上有些浪费，或者说过于占用数据库资源，于是进入第九个阶段。在这个阶段可能会形成的架构演变是实现数据读写分离，同时编写一些更为廉价的存储方案，例如 BigTable。

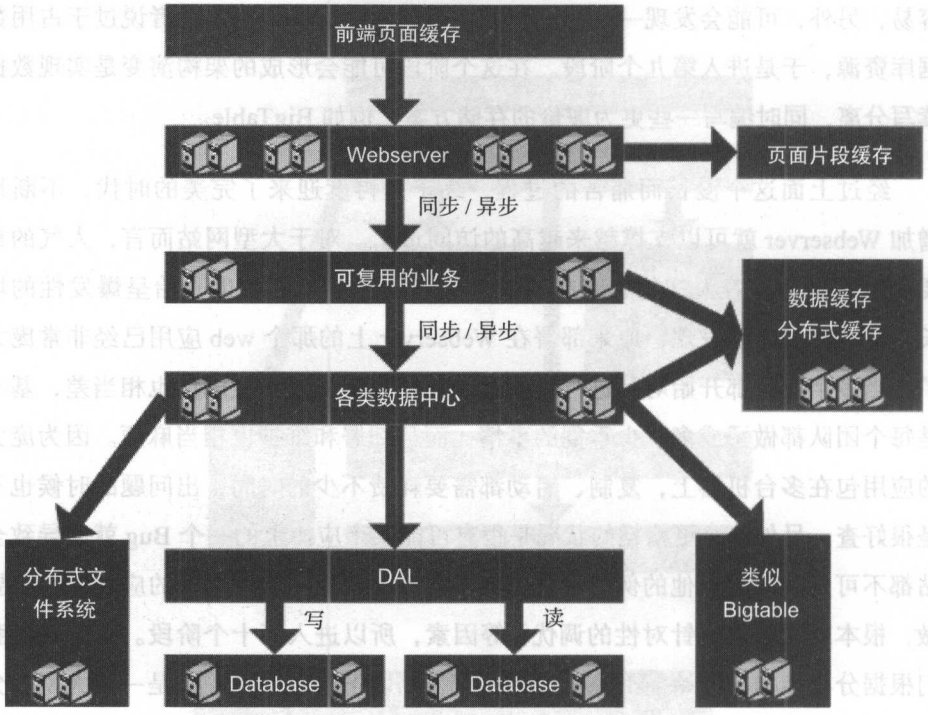
经过上面这个漫长而痛苦的过程，终于是再度迎来了完美的时代，不断地增加 Webserver 就可以支撑越来越高的访问量了。对于大型网站而言，人气的重要毋庸置疑，随着人气的越来越高，各种各样的功能需求也开始呈爆发性的增长。这个时候突然发现，原来部署在 Webserver 上的那个 web 应用已经非常庞大了，当多个团队都开始对其进行改动时，会相当不方便，复用性也相当差，基本是每个团队都做了或多或少重复的事情，而且部署和维护也相当麻烦。因为庞大的应用包在多台机器上，复制、启动都需要耗费不少的时间，出问题的时候也不是很好查。另外一个更糟糕的状况是很有可能某个应用上的一个 Bug 就会导致全站都不可用。还有其他的例如调优方案不准确（因为机器上部署的应用什么都要做，根本就无法进行针对性的调优）等因素，所以进入第十个阶段。这个阶段我们根据分析结构，开始痛下决心，将系统根据职责进行拆分，于是一个大型的分布式应用就诞生了。通常，这个过程需要耗费相当长的时间，因为会碰到很多的挑战：

1）拆成分布式后需要提供一个高性能、稳定的通信框架，并且需要支持多种不同的通信和远程调用方式。

2）将一个庞大的应用拆分需要耗费很长的时间，需要进行业务的整理和系统依赖关系的控制等。

3）如何运维（依赖管理、运行状况管理、错误追踪、调优、监控和报警等）好这个庞大的分布式应用。

经过这一步，系统的架构基本会进入相对稳定的阶段，同时也能开始采用大量的廉价机器来支撑巨大的访问量和数据量，可以结合这套架构以及这么多次演变过程吸取的经验来采用其他各种各样的方法来支撑起越来越高的访问量。



5.2.3 学习能力

如果你想进步、想要有所成绩，就需要持续学习、终身学习。

个人层面需要不断地输入，学习新的知识，保持对行业、领域内新技术的更新。看论文可以被认为是架构修炼的一种方式，因为很多论文写得比较严谨，也比较系统化，了解一个系统实现的细节对于架构方面的成长很有好处。

有一天我的一位同事找到我："周工，我看到你出的书了，能不能告诉我怎么提高自己的技术能力？"。我对他说："你每天 7 点起床，23 点睡觉，中间所有空闲时间都拿来学习、思考、总结技术问题，你就可以提高了。"这不是开玩笑，任何人想要提升自己的专业能力，有效、高效、有针对性地付出时间是最直接、最有效的办法。

架构师是一个充满挑战的职业，知识面的宽窄往往决定着一个架构师的架构

能力，你需要阅读大量的技术书籍，但是不要仅限于软件相关的书籍。要经常上技术论坛，一方面可以结交朋友，一方面可以拓展自己的知识面。公司的大小往往决定了所做的项目规模，一般的大项目不太可能直接总包给小公司去做，但这并不妨碍小公司可以分包到大项目的一部分。在做小项目的同时也可以积累丰富的经验。宽广的知识面对于一名出色的架构师来说是必不可少的。也许很多人对架构的理解还停留在设计模式、重构、SOA 等软件层面，然而这仅是非常基本的东西，架构师的脑子里不光需要知道让软件如何高效地运行，还需要知道如何去结合网络、存储，甚至一些文件系统的特性，比如 GFS、NFS、XFS、NTFS 等。而且架构师还需要知道一些编程语言的特性，如 C、C++、Java、PHP、Python、Lisp、JS 等。现在是一个混合编程的时代，只了解一种语言，即使再精通也会使你在架构系统的时候受到很大的局限。架构师需要对数据库技术有深刻的认识，因为现今是一个信息时代，大量的信息都是需要存储并检索的，数据库设计得不好，将会严重影响系统的性能，而这一点往往会被我们的设计人员忽略，他们只知道遵守那些范式而不会去结合数据的特性来设计数据库。

从一个程序员到架构师是一个很大的变化，架构师需要从大的方面考虑，而不只是考虑这个模块该用哪种设计模式去开发。总的来说，想要成为架构师，需要有耐心，不断学习，拓宽自己的视野，不要局限于自己眼前的项目，多关注开源技术，多关注热门技术社区的新动向。

5.2.4 创新思维

批判性思维是创新过程中最显著的特征，不求一鸣惊人，但求独善其身是程序员队伍中一种比较具有代表性的思想。但我们真正能做到独善其身吗？试想如果现有代码的可维护性很差，任何人都很难在有限的时间内写出高质量代码，最终很可能“随波逐流”。有时我们也会一不小心走到另一个极端，如果留意，我们总能在公司的茶水间里听到员工关于公司缺乏产品创新的批评。批评是有益的，但我们更需要批评过后的冷静思考和实际行动。创新无疑需要批判性思维，但前提是我们能将“批判”与“批判性思维”区别开，批判的目的只是为了达到

对某个事物的否定，而批判性思维则是通过否定来寻找更好的做事方法。单纯批判是消极的，而批判性思维是积极的。

对于软件研发团队，定义问题的能力直接影响其需求分析的质量，进而最终决定了产品的质量。一般认为软件工程师参与到需求分析与产品设计的机会并不是很多，他们是接收来自用户或者产品经理的需求规格，并且努力把功能列表转变为可以运行的程序。不幸的是，现实中软件生产过程要复杂得多。在开始阶段，写下来的需求只是冰山一角，未写下来的需求才是沉浸在水下的巨大冰块；即使写下来的、被用户所确认的需求也不能确保实现后一定能够让用户满意。

没有精神层面的支持，创新是无从谈起的，因为创新实在是一项艰巨的工作，概念的建立不是一蹴而就的，需要反复地推敲、对比、自我否定。问题的定义更是自我意识游走在理想与现实之间，尝试各种问题边界的可能，不断地进行权衡。人生贵在坚持批判性思维，而不是简单的批评与抱怨。所有这些行动都是很困难的，因为它们会产生巨大的认知摩擦。由此可见，对于一个致力于创新的程序员，能够建立并保持一个积极的人生态度才是最值得庆幸的事情。

我们可以将创新模型简单地归纳为以下 4 个核心要素：

- 创新需要挑战现状，即批判性思维。
- 批判性思维需要准确的问题定义。
- 问题的定义有赖于概念系统的建立。
- 在初始阶段，概念模型的正确与否是次要的，更为重要的是要有一个概念模型，然后这个概念模型可以随着认识的深入而不断演化。谁在幕后推动认识的发展？答案还是批判性思维。批判性思维有助于我们修正原有概念的不足或创建全新的概念，在思维的各个阶段都需要乐观的人生态度的支持。

通常我们的开发人员具有的也是模块化设计逻辑，程序完成后会打包并部署成一个个具体的应用。每个应用的格式依赖于相应的应用语言和框架。例如，Java 应用通常会被打包为 WAR 格式，部署在 Tomcat 或者 Jetty 上，而另外一些

Java 应用会被打包成自包含的 JAR 格式。同样，Rails 和 Node.js 会被打包成层级目录。这种应用开发风格很常见，也很易于调试，只需要简单运行此应用，用些工具链接 UI 就可以完成端到端测试。将打包好的应用复制到服务器端，通过在负载均衡器后端运行多个拷贝就可以轻松实现应用扩展。在早期，这类应用运行得都很好。但一个简单的应用会随着时间推移逐渐变大，几年后，这个小而简单的应用会变成了一个巨大的“怪物”。一旦你的应用变成一个庞大而又复杂的“怪物”，那开发团队肯定很痛苦。敏捷开发和部署举步维艰，其中最主要的问题就是这个应用太复杂，以至于任何单个开发者都不可能搞懂它。因此，修正 Bug 和正确地添加新功能将变得非常困难，并且很耗时。

一个应用越大，启动时间就会越长，大部分时间就要在等待中度过，生产效率会受到极大影响。

另外，一个应用在与其他模块发生资源冲突时，扩展将会非常困难。应用的另外一个问题是可靠性，当所有模块都运行在一个进程中，任何一个模块中的一个 Bug，比如内存泄露，都有可能弄垮整个进程。除此之外，因为所有应用实例都是唯一的，这个 Bug 将可能影响整个应用的可靠性。

如果采用微服务来处理结构模式则可以很好地解决上述问题。微服务架构的思想不是开发一个复杂巨大的应用，而是将应用分解为若干小的、互相连接的微服务。

5.3 常见问题分析

5.3.1 无从入手

现代的软件从业者，都受过良好的计算机和软件方面的教育。但是现代的计算机和软件方面的教育，基本上都是从科学研究领域脱离出来的，教育的主要目的理所当然的是为科学研究领域服务。而随着社会的发展，软件不断地渗透到不同的业务领域，涉及普通人生活的方方面面。以科学研究为目的的软件教育，和

日益深入人们生活的软件应用产生了很大的隔阂，以致很多计算机和软件专业毕业的学生进入企业工作后，总是感叹学校所学习的知识没实用价值，必须得重新学起，才能够达到企业的要求。

系统架构和上面的描述很像，貌似你看了很多的书，市面上也确实有很多例如“分布式系统架构”“微服务架构”等跟随潮流的书籍，但是看完后只停留在会采用一些开源框架进行整体框架搭建（注意，我说的是搭建，而不是设计）。确实是搭建，你所拥有的能力就好像小孩子搭积木，只会采用固定套路，或者学得差点的连固定套路都不会，这样对你的个人发展其实没有多大好处，这也是为什么很多程序员在完成了程序员到架构师的转型后，没过多久就转为纯管理，或者彻底离开了技术界。因为他们从来没有大彻大悟地理解系统架构。

如果你发现一个人画出来的总体架构图有点奇怪，没有区分服务端、客户端，这时候你可以猜测他并没有真正理解什么是系统架构，他更多的是以模块方式进行切割，而不是从架构设计角度思考。出现无从下手或者随意下手的情况，真正的原因都是不理解，建议大家从软件系统架构基础类的书学起，不要一开始就看基于潮流框架的架构书。

5.3.2 轻视设计

一般来讲，职场新兵在设计业务系统的时候，容易出以下几个问题：

- 需求沟通完，马上就开始设计数据表结构，就想着有哪几张表、要建哪些索引、有哪些唯一键、如何保证事务、如何设计触发器等。出现这类问题很正常，毕竟在学校学的都是理论，授课的老师很多没有太多的工程经验，也指导不了你的设计。
- 要他设计一个系统，完全摸不着头脑，一提笔就是写代码，想到哪里写到哪里，没有一丝设计。出现这类问题的同学，一般都是在学校没好好学习的同学，或者是非计算机专业的同学。
- 而有一些工作经验的人，做出来的设计也是一团糟，不是考虑不周全，就

是画的图别人看不懂。这类人一般是平时不喜欢看书学习，或者是之前刚工作的时候没有有经验的人指导。

架构师需要参与到项目开发中，这样团队成员就不会出现轻视设计的情况。架构师参与项目需要分阶段。

	活动
需求	架构师无须参与到需求阶段，这是一个常见的错误观念。事实上，这正是架构师能够理解利益相关者所认为的、所谓的软件“质量属性”的关键
分析和设计	在这一阶段，架构师提供系统的概要设计，确定软件要素、接口及它们之间的关系。在这一阶段，还描述了这些要素的部署
执行和开发	实际的开发和编码都是在这个阶段中进行的。在这个阶段中，需要对软件体系结构进行确认和修改，这可能会导致对设计的更改、对项目时间安排的更改
测试和质量保证	在这一阶段，需要验证软件的质量属性（可伸缩性、性能特征等）
部署	在这一阶段，IT 架构师需要仔细检查平台上软件应用程序的最终部署

5.3.3 技术优先

技术本身是没有好坏的。比如最早的计算机软件是单机运行的，而后逐渐发生了架构拆分，最终形成了 C/S 的架构，随着互联网技术的发展又成了 B/S，其实这都是换汤不换药，Browser 还是一个 Client。但是因为 Browser 的普适性，又做了一个架构拆分，成了一个通用的 Client，做到了很多 C/S 时代做不到的事情。互联网时代的技术人员对 C/S 架构不太认可。到了 App 时代，又回到了 C/S 时代的做法，因为 HTML 5 响应慢，无法适应手机客户端。所以说，随着人们问题的不断变化，会有不同的技术周期出现，技术有自己的生命周期。没有永远最好的技术，也没有永远最差的技术，问题总是在不断发生变化的，对于这一点，架构师要有清醒的认识。

我其实不太喜欢面试培训班出来的技术人员，他们可以滔滔不绝地和你描述各种框架的使用，细微到某一个参数，每遇到这种场景，我真想用英语说“I don’t care。”并不是有偏见，而是我真的不在乎你懂不懂参数的配置，我要问的是计算机科学基础知识，可以聊算法、聊数据结构、聊设计模式、聊你参与过的系统架构设计、聊内存管理机制和垃圾回收机制、聊你对于技术的情怀。框架的

选择，实质上是对技术可行性的选择，这又需要符合当前的业务形态，所以，没有哪一个技术或者框架是最好的，只有最适合你的产品需求的，才是最好的。

想做好技术的使用者并不容易。架构师必须深入了解不同的技术，知道这些技术的强项和弱点，懂得合适取舍。

技术人员如果要成为架构师，就必须跳出技术的视角，换一个角度去看技术。要把时间花在研究生命周期规律和业务的增长上，花在选择合适的技术上，而不只是追求新潮的或自己喜欢的技术。在很多人看来，特别是软件工程师，架构和技术是完全等同的。多学会几种技术，就觉得可以做架构师了，或者学会的技术越多，就觉得自己的架构水平越高。对于别的架构师也会采用同样的标准来评判。你要知道，任何技术都是为了解决某种问题而存在的，学会了很多技术，并不代表能够利用这些技术来解决问题，学会的技术多只是解决问题的手段多了一些而已。

架构师需要很冷静、很平等地对待所有的技术。对于软件架构师来说，他们要深刻地领会软件开发生命周期和软件运行生命周期，以及业务的生命周期，并把它们合理地组织、结合在一起，能够通过软件的快速增长来支撑和顺应业务的增长。这些是软件架构师的核心能力。为达到这些目标，无论是新的还是旧的技术、先进的还是落后的技术，只要场景合适，他们就会采纳。因为架构师是技术的使用者，他们把技术当作解决增长问题的手段和工具，而不会被技术束缚住。

5.3.4 业务困境

业务和架构都是处理人的问题，而技术人员最讨厌处理的就是人的问题，内心厌恶，却又无法逃避。因为这个排斥的心理，工作中始终想避开和人有关系的地方。因此在做技术之前，还需要做一些准备工作，用来连接现实生活，让大家知道处理人的问题并不可怕。建立了这个相关性，每个人就都可以自行思考。

其实对于做软件的技术人员而言，技术也可以分为两部分：一部分是软件技术，另一部分是业务技术。当前软件行业所说的技术，基本上都是指软件技术和

计算机相关的技术，也就是软件的访问生命周期所涉及的技术，比如服务、存储等相关技术。软件技术大部分集中在软件的访问生命周期部分。业务技术这一块，软件工程则较少涉及，这部分主要隐藏在业务逻辑中。因此在谈技术时，只谈软件技术是远远不够的。不管是软件技术还是业务技术，均来源于对现实生活问题的解决，现实生活才是架构师真正的营养来源。

为什么软件工程师会有时间恐惧和压力呢？原因是他们把按时完成自己的工作当成了自己的最大利益。人对时间的压力是与生俱来的，对业务的不了解也会导致他们没有太大的把握。这一问题在其他行业的表现并不明显，毕竟在其他行业，架构师主要处理的是本行业的问题，对业务比较熟悉。软件行业则较特殊，通常是以业务的问题是否解决为判断标准，是在解决另一个行业的问题，这一点提高了对软件架构师的要求。这就要求软件架构师把完成业务的工作当成自己的最大利益，深入到业务中去。随着对业务的熟悉，对时间的恐惧才会慢慢消失。对业务领域理解得越深入，就越知道如何去发现问题，慢慢就成为业务专家了。只有做到这一点，才能在业务领域建立自信，成为一个合格的软件架构师。

还有另一种很普遍的现象，做技术的软件工程师往往看不上业务，觉得技术更高端，而业务太平凡、太低端，并且业务人员总是给技术挖坑。而业务人员则觉得做技术的眼光高，但总是理解有偏差。技术人员往往对业务一知半解，业务问题总是解决得不圆满，但业务人员对此又无可奈何，因为自己不懂技术。做架构的人必须亲身体验业务，感受业务，才可能真正认识业务的个性，真正认识业务所面临的问题。在理解业务个性的基础上，才能够谈共性。否则这个抽象就像是无根之木，一旦局限于个人的主观认识，很难长大。

如果软件工程师对业务一知半解的话，软件是不可能写好的。在软件开发的整个过程中，参与其中的所有角色，都应该以软件工程师为核心，帮助软件工程师理解业务，让软件工程师成为业务专家。这听起来似乎又回到了科学研究领域的研究人员采用软件的方式，只不过研究院是先学业务再学软件，而软件工程师是先学软件再学业务，顺序不同而已。只有软件工程师成为业务专家，写出来的软件才是靠谱的。

如果业务简单，流量足够小，几个软件工程师就可以了，并不需要其他角色参与。比如一个创业公司，可能几个软件工程师就搞定了。一旦业务复杂度比较高了，或者流量达到一定规模，就需要很好地组织软件工程师的分工。

5.3.5 专职架构师

如果软件架构师不具备组织架构上的权利，那么架构师的设计慢慢就会被弱化，无法落地。软件工程师或者团队领导会按照自己的想法擅自修改或者重构架构，尤其是软件工程师容易不计代价地采用流行的技术或者自己喜欢的技术，甚至会和架构师产生冲突。这种情况下，对于软件架构师内心的冲击会很大，他们往往会选择变成软件工程师，专注于深入软件实现过程，而不顾及软件和业务的增长。

技术人员如果要成为架构师，必须跳出技术的视角，换一个角度去看技术。要把时间花在研究生命周期规律和业务的增长上，花在选择合适的技术上，而不是漫无目的地追求新潮的或是自己喜欢的技术。

对于从软件工程师成长出来的软件架构师，要真正成长为一个架构师，首先需要克服的就是时间困境。从软件工程师的角度出发，首先需要面对的就是可以用自己的软件技术能力去解决业务问题。这些业务对于软件工程师来说，是另一个行业的内容，因此软件工程师对于业务往往一知半解，也没有太大的动力去研究，他们主要专注于完成自己手头的编码工作。如果一个人在工作中，只是致力于完成自己的工作，以做好自己的工作为主要目标，那么用自己行业的知识去理解另一个行业，就很容易出现沟通的困境，造成很多理解上的困难。

当软件工程师需要帮助别人解决问题，并且按时、按需解决业务问题已经成为他们自己的问题的时候，软件工程师就有了时间的压力，潜意识里会自然而然地对时间产生恐惧。这个恐惧会想方设法地推动软件工程师采用各种手段，如加班，以便及时地完成工作。要想成为架构师，必须超越对时间的恐惧，看清楚需要解决问题的主体是业务人员，而不是自己。即需要解决的问题是另一个行业的问题，自己

是在帮助业务人员解决问题。这也是软件特别的地方。如果只顾着完成自己的工作，而业务的问题没有解决，那么很可能需要返工，从而导致软件工程师投入更多的时间。工作是否完成是由业务人员决定的，而不是软件工程师自己。

5.3.6 一步到位思想

可不可以把软件架构一步到位地做好？在当前的技术水平下，这基本上是不可能的。一方面在流量不大的时候，做太复杂的拆分会浪费大量时间和不必要的成本；另一方面是因为流量是外在的因素，有经验的人可以做出部分预判，但也没有办法完全预测流量会在哪个地方增长。所以既没有必要一步到位，也不可能一步到位。只能结合运维和运营，及时地探索流量的增长，在流量达到瓶颈之前，根据业务的规律进行合理的拆分，帮助快速地应对增长。这不是架构演进，因为此时业务并没有发生变化，只是流量增长了，把这叫作软件的长大可能会更合适一点。

5.3.7 系统过度设计问题

设计的方案如果不结合目前的实际情况，考虑得太复杂，实现起来也会特别复杂，和造轮子一样，也存在同样的浪费，其实过度设计倒还好，就怕错误设计。比如觉得 MySQL 性能不好，要加一层 Memcached 缓存，最后设计并上线使用后发现，缓存命中率非常低，白白浪费了大量服务器，损耗了性能，并增加了系统的复杂性。

2012 年支付宝遇到了健康检查系统问题。和所有系统的自我保护一样，这个健康检查系统会定时扫描线上机器，根据机器应答返回时间判断是否正常，将超时严重的机器从应用列表中剔除。在双 11 的流量之下，几乎所有机器都发生了响应过慢的情况，然后大部分机器都被剔除了出去。发现问题之后，支付宝快速下线了这个系统，支付成功率才重新回到了正常值。

5.4 其他

5.4.1 架构和树的关系

我们每次谈到架构，都需要谈论树状结构，那么树状结构有什么好处呢？树状结构特别适合增长。树状的增长，成本方面最低、沟通的路径也没有显著的增长，带来的效果最好。树的结构也保证了分支的能量汇集到树干，保障了整棵树的生长。而架构恰恰是为了应对增长的，自然而然架构都是树状的。

谈到树，就必须要提到分层。分层实际上是架构树状拆分的结果，所以当我们采用分层的时候，内心先要有树的概念。如果我们直接采用分层的方式设计，最后就会违背树的原则，导致很多复杂的问题发生，从而影响到系统的长大，例如跨层访问形成了图，这就违反了树的原则。

架构的拆分是顺应业务生命周期规律的一种拆分，这个拆分始终是围绕着业务的核心生命周期进行的，结果只有一个：无论如何拆分，它依然是树。

注意，很多人一谈架构就会提到分层，分层的前提是把一个生命周期的连续过程拆分成一棵树，因为有了树才会有层的出现。

5.4.2 为什么会有架构

如果业务足够简单，用户流量足够小，时间要求也不紧迫，那么一个人或者一台机器慢慢实现就可以了。这种情况一般不需要讨论架构的问题，因为不需要并行，即没有时间压力，也没有产出的压力。此时对应的业务通常比较简单，对应的业务组织架构也会比较简单。

当软件对应的业务组织架构越来越复杂，或者当访问的流量越来越大时，就会产生时间的压力，即在相同的时间内需要完成更多的产出。软件会拆分得越来越复杂，机器也会越来越多，甚至同一个软件也需要拆分为多个组件，需要多人合作完成编写。但是业务的核心生命周期，即业务的核心建模不会有任何的变化，还是完成同样的事情。唯一的区别就是会拆分出很多的非核心生命周期，它们以

核心生命周期为根，不断地增长，最终形成一个树状架构。生成的架构如下：

1）在软件开发生命周期中参与的人员越来越多，进而形成软件开发团队的组织架构。因为软件代码开发的过程是一个生命周期，必须严格按照时间有序推进，根据开发的核心生命周期进行拆分，形成很多个非核心生命周期。按照软件开发核心生命周期的流程，把不同的非核心生命周期串联起来，使得这些非核心生命周期能够并行开展工作，这就是软件工程。所形成的就是软件开发团队的组织架构。

a）为了让业务在软件中实现并落地，便于用户对软件的访问，需要不同技巧的人同时工作，并对代码进行切分，形成代码的架构。切分的原则就是根据用户对软件访问的生命周期，识别出核心生命周期和非核心生命周期，形成树状架构。从分层的角度进行思考的时候，千万注意不要破坏树的架构。有树才会有分层，有分层却不一定有树。当这些工作由一个人来完成的时候，由于没有明确的分工，不一定会有代码架构，所以代码往往会比较混乱。

b）为了完成业务的工作，需要把识别出来的业务架构和支撑业务的组织架构，以及业务运行的核心生命周期流程，用代码表述出来。

2）在软件运行生命周期中，当软件的流量越来越大时，慢慢就会超出软件所在机器的负荷。这个时候就必须增加机器，软件所部署的机器也会根据用户访问的生命周期，按照树状的结构开始拆分，所形成的是硬件部署树状架构和软件部署树状架构。

5.4.3 业务、技术、架构三者关系

业务、架构和技术之间是共生的关系，而不是互斥的关系。

技术人员很多时候所关心的技术，和业务的主要目标往往不是直接对应的。由于业务和技术属于两个不同的树，也就是说有两个不同的根节点，因此只有沟通才能解决问题。

技术总是在人类对业务目标要求不断提高的情况下产生，其目的是为了获取

更大的利益。所以，技术是为了解决业务问题而产生的，没有了业务，技术也就没有了存在的前提；有了更好的技术后，效率较差的技术，就会慢慢被淘汰，进而消失，一切都遵从人类的利益诉求。

不同技术之间有两种关系：

1）在解决同一个业务的前提下，更高效、更低成本的技术会淘汰低效、高成本的技术。这是由人类利益诉求所决定的。

2）通常刚开始解决核心业务问题的核心技术的效率是比较低的，只是把不可能变成了可能。慢慢就会有提高的需求出现，改进技术的要求就会变得很迫切。技术所解决的业务生命周期慢慢就会开始发生拆分。非核心生命周期分离出去后，要么使用现在的技术来实现，要么形成新的技术，服务于更广泛的业务。

在业务生命周期拆分之前，技术是由一个主体串行地执行并工作的，这就是效率低下的原因。从业务生命周期中把非核心生命周期分离出去之后，非核心生命周期会形成新的技术，由不同的主体来执行。新的技术可以独立地与核心技术并行工作，提高原有技术的效率。因为要解决的核心业务问题并没有发生改变，所以拆分所形成的是一个树状的架构。

也就是说，先有业务问题，才会有技术来解决业务问题。而业务的长大要求，提高了对技术的要求，导致了对业务生命周期的拆分，以并行的方式提升效率，形成了架构，也形成了新的技术。所以在这三者的关系里：业务是核心，技术是解决业务问题的工作，而架构是让业务长大的组织方法。架构需要用技术来实现拆分，技术需要架构来合理组织，以提升效率。软件和业务最终是要合体的。

5.4.4 架构拆分要点

需要遵循拆分的原则如下：

1）如果必须要生命周期的主体在连续时间内持续执行，不能够被打断并更换生命周期主体，那么被拆分的生命周期，不能切分出去。

2）每个生命周期的负责人对所负责生命周期的权利和义务必须是平等的。

3）切分出来的生命周期不应该超出一个自然人的负载。

4）切分是内部活动，内部无论怎么切，对整个系统的外部都应该是透明的。

架构切分的过程就是建模的过程。在早期流量不大的时候，往往一个人会同时兼很多工作，一个系统往往也会做很多的事情。在流量增大后系统很快就达到瓶颈，这个时候就不得不进行拆分了。要做拆分就必须要去识别核心生命周期和非核心生命周期，把非核心生命周期切分出来。切分出来的不同子生命周期就形成了不同的概念。所以每个概念背后都是一个生命周期，每个生命周期都是一个模型。核心生命周期模型把所有的模型通过树组织起来，形成了一个新的模型。

架构切分的结果最终都会体现在组织架构上，因为架构的切分是对人利益的重新分配。另一方面，架构切分需要组织架构来保障实施。要为负担重的相关利益人减轻职责和权力，要为负担轻的需增加职责和权力，这里反复强调职责和权力，它们本质上应该是需要对等的。如果所有人负担都很重，就要增加人，从而形成新的架构切分；或者引进新的技术，提升大家的生产力，以形成新的架构切分。所以进行架构切分的时候，往往也是组织长大的时候。

一个好的架构拆分会形成一棵树，慢慢会长大成一片森林。每棵树在这个森林里都能够获得所需要的养分，有自己的空间。每棵树的内在是平和的、舒展的，遵循自己的生命周期规律，顺其自然地长大，一派欣欣向荣的景象。这就是架构的魅力所在。

5.4.5 企业应用架构演进

不论是传统企业，还是互联网公司，发展到一定阶段，都需要一整套体系化的应用架构来支撑其运转。良好的、合理的应用架构可以支持企业高效开展业务，控制经营风险，而混乱的、不合理的应用架构则会限制企业的快速发展，成为企业增长与变革的瓶颈。

完整的企业架构（Enterprise Architecture，EA）分析构建，包括业务架构、应用架构、技术架构、数据架构，这里着重介绍应用架构。

1. 什么是企业应用架构

从开发模式上看，传统企业更加倾向于瀑布式软件开发，对研发、运维流程的管理更加严谨，因为传统企业业务变化慢，对系统的每一次调整改造都非常慎重，而互联网公司业务变化调整快，必然要求软件开发时效性高，对软件设计的严谨性做出一些让步，因为很有可能发生的情况是，软件还没有开发完毕，业务或流程已经再次发生变化。

企业 IT 架构的设计不仅仅是注重某一个系统能力，更需要给企业一个可进化的、可持续发展、不断创新的平台。业务持续变更将成为“新常态”。架构师们不仅需要满足企业 IT 中高可靠、高安全、一致性、合规性要求，还需要满足创新 IT 所需要的灵活、快速、伸缩的挑战。

2. 企业信息管理模型及案例

企业的信息化管理有很多标准模型，例如 COBIT、ITIL、CMMI、ISO27001，这些覆盖了开发管理、服务管理、数据管理等方方面面的规范和标准。很多成熟的互联网企业也会执行使用这些标准来提升 IT 能力对企业管理效率的提升和经营风险的控制。

当你的企业还是一个小作坊的时候，比如经营一家小超市，这时候你只需要用 excel 表格就能管理好自己的账目，因为此时无非是类似于进销存系统的买进价格、卖出价格这类统计。你也可以采用科学的数据表格管理，记录门店的所有采购入库和销售数据，这会让你的经营变得井井有条。通过这些原始数据，你可以准确地管理库存，计算利润，掌握畅销品和滞销品，还能通过数据透视表制作销售日报和月报。这已经是一个管理软件的雏形了。

接着由于经营得不错，你的小店逐渐扩大了规模，吞并了旁边的其他小店，有了连锁店的感觉。这时候，你发现管理有问题了，需要引入 ERP 系统。你选

择了一套轻量级的ERP，并且只购买了其中的几个核心模块，这样既可以控制成本，又可以让你经营的软件设备升级。

而后连锁店生意不错，为了更加准确地理解、认识你的客户，同时也为了能够拉近你和客户的距离，你打算通过CRM软件进行更加科学的客户管理。核心的客户信息资产模块都在CRM中实现，其中内置了营销模块、消息推送服务MSG模块，包括SMS、EDM（Email Direct Marketing）和微信消息推送。CRM主要聚焦于客户资料的管理和营销服务，ERP主要聚焦于超市的进销存及财务业务。这时候就有了应用架构设计的概念，公众号、ERP和CRM每个系统都为了解决某一大类的业务问题而存在，有各自清晰的定位、分工和目标用户，每个系统相对独立又互有关联；内置若干模块，每个模块都是为了解决某一大类业务问题下的某一小类问题而设计。

业务进展很顺利，你已经开了五家中型连锁超市了，员工数量达到了几百人。公司走上了正轨，标准化的管理分工已经成型，不同职能单元各司其职。为了有效管理团队，并且让内部流程更加顺畅，你邀请专业的IT咨询公司来帮你重新梳理公司的业务目标、组织架构、运营流程，通过引入OA、HRM以及重构ERP等手段，对不合理的制度、低效的流程进行了改造。公司成立了信息技术部，其中项目部配合咨询公司及软件外包公司进行系统改造或实施新系统；运维部负责保证服务器、网络的稳定。理解数据对公司发展非常重要，所有的管理决策都应该基于对数据的分析和判断，因此你应邀请咨询公司帮你强化公司的数据分析能力。咨询顾问建议你实施数据仓库（Data Warehouse）和BI（Business Intelligence）项目，原因有几点：

1）ERP系统和CRM系统都有报表模块，但两个系统的数据相互孤立，不利于整合分析；

2）业务系统的底层数据结构并不适合做复杂的数据分析，常见的多维分析更需要一套数据仓库常用的星形数据结构和雪花型数据结构；

3）成熟的BI软件套件可以让你的报表分析与多维数据探查更轻松，其中的仪表盘更能够让你轻松掌控公司全局的核心指标变化；

4）企业经营中很常见的一个问题就是经营分析指标统计口径太多，造成管理混乱和沟通障碍，除了在管理上要规范公司级指标的定义外，也需要一套底层数据架构，以消除上游各个异构系统的孤岛和屏障，便于统一管理、汇总数据并进行指标计算。

咨询顾问建议，虽然目前公司的业务系统还没有到非常复杂的阶段，但数据仓库可以帮助企业更快速高效准确地理解、捕获、使用数据，做好基础建设工作，培养员工的数据分析意识和方法，通过数据来进行决策。随着业务的拓展和系统复杂性的提升，数据仓库的存在价值将越来越明显。

由于公司经营良好，很多商品可以从供应商处拿到很好的价格，经过供应商授权，公司决定开展 2B 业务，成立了大客户销售部，公司将作为供应商的 B 端渠道，挖掘企业客户。为了让销售工作高效展开，对销售人员进行严格的过程管理，同时也为了保留客户资料，避免销售独占客户资源，根据 CTO 建议，公司决定实施操作型 OCRM（Operating CRM）项目。同时由于各部门经常出现个性化的软件开发诉求，软件外包维护的成本高、效率低，公司决定招聘研发团队，用自己的队伍进行软件的二次开发。

在设计 OCRM 系统时。CTO 面临两个选择。

方案一：新做一套独立于现有 CRM 的 OCRM 系统。

优点：OCRM 系统已有成熟的软件可以选择，无须从头开发；两个系统边界清晰，分工明确，便于未来各自的发展与演变。

缺点：应用架构会略复杂，需要将原有的 CRM 和 OCRM 做数据打通，对原有的客户模型做升级。

方案二：在原有的 CRM 基础上开发新模块。

优点：新开发的模块完全基于公司业务流程和模式设计，适配程度高。

缺点：新开发模块成本高、速度慢，系统边界模糊，导致以后维护升级时模

块管理混乱。

综合评估两套方案实现的成本和速度，考虑到对未来业务变化的灵活支持，同时为了避免影响核心 CRM 业务的稳定性，CTO 决定采用方案一，让两个系统各自聚焦，互相独立，边界清晰，虽然无形中增加了公司应用架构的复杂性，但可以快速实施支持当前的紧迫业务，并灵活应对未来公司的销售业务变化。

至此，我们已经绘制出一套一般企业的简化版应用架构图，以及一张常见的组织架构图。可以看到，应用系统的建设是根据业务的发展而逐步完善的，每个系统都有独立存在的意义和价值。

3. 给架构师的建议

不论是架构师、产品线负责人或某个系统的产品负责人，都要有架构设计的理念和知识，尤其是后端产品经理，必须充分理解企业应用架构的基本概念。这里给出一些针对应用架构设计的建议。

1）系统定位和边界要清晰，对应的业务定位和边界要清晰。一套应用系统的存在，是为了解决某一类业务问题，对应某一个业务板块。如果业务板块或业务单元定义模糊，也会导致对应的应用系统定位混乱。

2）系统要实现松耦合、高内聚[⊖]。对外界系统要透明、简单、易理解，与外部系统的接口要简明、扼要、灵活。内部模块高度聚合，粒度越细越不可拆解。

3）易变的、尝试中的新业务要避免影响现有业务的稳定性。对新业务的支持，可以考虑新建独立微小型应用系统，以便避免改造成熟核心系统，影响其稳定性和健壮性。

4）系统之间数据要实现单向流转。系统之间尽量保证单向数据流转，确保数据流可回溯性、一致性和可追溯性。混乱的数据流转管理会造成应用架构管理的灾难。

5）架构设计核心目标是支持业务，有时候不合理的存在是合理的。应用架

⊖ 指某个生命周期的变化是积累在一个生命周期主体上的，而不是分散在不同的主体上。

构存在的首要目标是支持业务，很多成长型企业或初创公司面对生存的压力，不能为了保证架构的合理性而拖延系统实施速度而导致企业错过发展时机。这种情况在互联网型企业更为常见。业务还在试错期，系统需要尽快保证支持业务试错，如果一上来就谈论整体架构的合理性，很可能花费巨大成本实现了合理架构后，新业务已经取消或失败。优秀的架构师和 CTO 要懂得在合理架构设计和灵活多变的业务发展之间做出智慧的权衡取舍。

对于 CTO 或公司架构师而言，要保证整体企业应用架构的合理性，只要大框架合理，局部的偏差可以忽略，修正的成本也比较小。如果大框架有偏差，修正的代价会非常高。对于产品线负责人而言，要保证局部框架的合理性，避免出现由于设计不合理造成的返工和补救工作。很多时候架构师或产品线负责人要做出判断，是做一套新系统还是修改老系统，若是前者则新系统如何定位，若是后者则老系统如何调整定位。另外还考虑：数据如何流转，系统之间如何关联，底层数据如何打通；是否要复用其他系统模块，是否要将某些模块抽象化、服务化、平台化。对于产品经理，要在系统级别的粒度做出类似问题的判断，能够识别出可能存在的系统演变风险，及时升级控制不了的问题，进而避免做出错误决策。

5.4.6 架构师分类

在软件生命周期拆分树中，软件架构师本身的业务也会涉及架构拆分，拆分出不同位置的架构师就有了各自不同的架构范围：有负责部署架构的架构师，如系统架构师；有负责数据架构的架构师，如数据架构师；有负责应用代码编写架构的架构师，如应用架构师。

1. 硬件架构师

硬件架构师的工作职责包括：

- 自主服务器硬件系统的架构设计及研发工作；
- 承担从业务向技术转换的桥梁作用；
- 协助制定项目计划和控制项目进度；

- 辅助并督导上游 ODM/OEM 开展设计工作；
- 负责建立适合需求的服务器硬件质量标准及检测流程体系；
- 负责组织重大项目技术研究和攻关工作；
- 负责带领公司内部员工研究与项目相关的新技术。

硬件架构师负责辅助并指导基于需求的硬件架构设计工作，需要针对不同的业务需求选择合适的技术路线，制定最优的技术解决方案。

2. 数据架构师

数据架构师负责制定数据标准、应用标准、运维标准，设计数据标准和模型管理流程，整理数据需求并为建模人员提供支持。其工作职责可以包括以下内容：

- 研究与跟踪大数据新技术发展方向，主持制定大数据平台技术发展战略规划；
- 负责 ODS 以及数据仓库逻辑模型、物理模型的分析与设计，并与团队共同制定核心模型的演进路线；
- 负责数据仓库的业务探索（Business Discovery）及信息探索（Information Discovery）的工作；
- 负责数据平台的设计、开发、维护与优化，不断创新，满足上层数据运营体系各项需求；
- 审核数据平台项目总体技术方案，对各项目进行质量评估；
- 参与应用分析系统的系统分析、设计以及实施工作。

3. 应用架构师

应用架构是从业务到 IT 转换的第一步。应用是对（系统）能力的分组，实现业务功能并且管理数据。应用架构描述这些应用之间的结构和逻辑关系，应用架构设计的一条重要原则是技术中立，也就是说应用最好是技术无关的。同样的，技术架构是描述支撑应用的技术实现的架构，比如技术标准、平台架构的设计、硬件基础设施的架构设计（比如网络拓扑）、系统请求响应的处理过程等。应用架构师必须是业务专家，直接承接从业务到 IT 的转换。应用架构师是否能做技术

协调管理就要看分工和能力了。

应用架构师的职责不同于网络架构师，网络架构师“不求有功，但求无过”。应用架构师架构的目的是直接为企业创造价值，为企业发展提供动力，提高管理水平。因此，应用架构师在进行架构时要充分理解用户的需求，准确定义用户的需求，通过与用户交流从而为他们开发综合的解决方案。应用架构师在架构过程中会在不同程度上影响人员、流程和技术，一旦他开始进行架构的构建，就是在创造一种环境和条件，让流程更加有效、高速和符合用户的需求。如果说 CIO 的工作是一个从无到有的过程的话，那么应用架构师就是一个从有到优的过程，要通过统一的规划管理使企业的 IT 资源发挥更大的价值。

4. 系统架构师

系统架构师是一个最终确认和评估系统需求、给出开发规范、搭建系统实现的核心框架、澄清技术细节、扫清主要难点的技术人员，主要着眼于系统的“技术实现”。因此他应该是特定的开发平台、语言、工具的大师，对常见应用场景能马上给出最恰当的解决方案，同时要对所属的开发团队有足够的了解，能够评估自己的团队实现特定的功能需求需要的代价。系统架构师负责设计系统整体架构，从需求到设计的每个细节都要考虑到，把握整个项目，使设计的项目尽量效率高、开发容易、维护方便、升级简单等。

系统架构师负责提供运营支撑软件应用的信息系统的结构设计，一般以满足各种非功能性需求或运营性需求为设计目标（如安全性、可伸缩性、可互操作性等）。

5.4.7 如何拆分

1. 代码拆分

在代码层面，要对业务代码和访问代码做好架构拆分。业务代码是软件访问生命周期的核心，软件运行的拆分受限于软件代码的拆分。因此要确保业务代码符合业务的生命周期，使得业务生命周期活动的结果积累在生命周期的主体上，

也就是内聚性，避免散落到访问代码中。这样软件的拆分就不会有太大的问题。

2. 架构拆分

系统发布之后，需要由一套统一的组件来解决分布式带来的共性技术问题。比如提供服务的发展机制、提供服务的分组路由机制、同机房优先机制等，这些能力沉淀在了一个叫 HSF 的框架里。为了解决单库性能瓶颈问题，使用了分库分表的技术，这个技术被沉淀在了 TDDL 框架上面。为了解决分布式事务的性能问题，把原本一个事务里的工作拆成了异步执行，同时必须保证最终数据的一致性，我们采用了消息发布订阅的方式来解决，这个消息框架就是 Notify。

通过采用分布式中间件，有效地解决了应用分布式后带来的技术扩展性问题，同时让整个系统的技术架构变得如当初一样简单。如果系统计算能力不够，基本上能做到只增加服务器即可。共享服务层和分布式中间件使频繁的业务变化被封闭在了一个适合的系统层，同时技术的变化也被隔离在了一个合适的范围。

5.4.8 一些案例

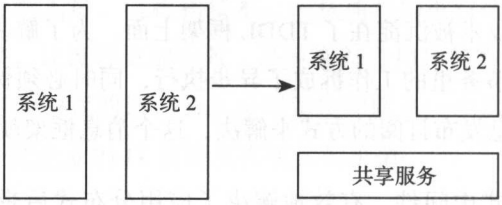
要想使用开源技术，就要理解开源技术都有哪些特定的使用场景，采用之前必须了解这些技术原本是用来解决谁的问题的。如果不先搞清楚就使用这些新技术，很有可能会导致更高的研发成本。小流量的公司采用针对大流量的新技术，仅仅维护和监控就足够让人头疼。如果环境配置跟不上，则技术是无法发挥作用的。新技术最重要的配置就是人的配置，以及人的观念的配置。

作为开源技术的使用者，要掌握作者的理念，一般要先从理解作者面对的问题入手，也就是从业务入手，分析作者是如何拆分业务生命周期的。从这个角度来说，看代码和看书没有区别。

1. 数据共享

为了解决业务扩展性问题，首先需要建立共享服务层，把公共的业务元素抽离出来形成共享服务。

在2008年，天猫和淘宝网是互相独立的两套系统，天猫有自己独立的成员、商品、交易、店铺、优惠积分等系统，唯一和淘宝共享的是会员数据。在天猫运行半年之后，由于数据和系统的独立，很难方便快速地借力淘宝网的大流量，这不符合互联网时代业务快速变化的特性，所以需要彻底打通淘宝网和天猫的数据和系统。



“房子千奇百怪，但是砖头都是一样的。”利用共享服务层解决了业务扩展性问题，好处是新构建一个业务市场变得非常容易和迅速，同时任何数据结构的变化只需在一个地方改动。带来的挑战是系统采用分布式之后研发要关注分布式本身。我们希望研发人员仍然像之前开发单机版的软件一样开发系统，把分布式的问题控制在一些通用的组件里面。这就需要引入分布式问题的中间件技术。

2. 共享服务

共享服务层的建立为横向业务提供了统一的数据和服务收口，比如手机、安全、商家服务这三个横向的业务就非常依赖共享服务。

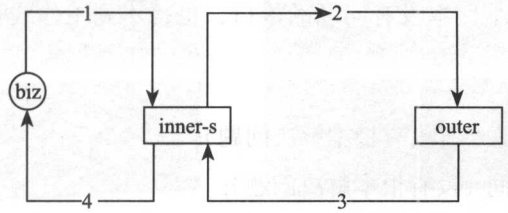
各个共享服务之间形成了比较好的隔离，保障了各个共享服务独立的发展空间，各个共享服务既互相关联，同时又互相独立。架构的域之间低耦合、高内聚。因为隔离做得比较好，没有业务之间的复杂交错，所以各个业务领域发展创新不受限。最佳案例是早期支付逐步发展成为支付宝，物流域逐步发展成为菜鸟物流。

3. 服务宕机

很多时候，业务需要跨公网调用一个第三方服务提供的接口，为了避免每个调用方都依赖于第三方服务，往往会抽象出一个服务：

- 解除调用方与第三方接口的耦合；
- 当第三方的接口变动时，只针对服务进行修改，而不是所有调用方均修改。

此时接口调用流程是什么样的呢？



由上图所知：

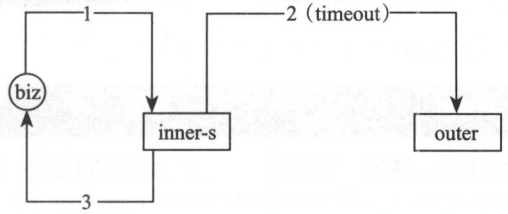
1）业务调用方调用内部 service；

2）内部 service 跨公网调用第三方接口；

3）第三方接口返回结果给内部 service；

4）内部 service 返回结果给业务调用方。

这个过程存在什么潜在的大坑呢？



内部服务可能对上游业务提供了很多服务接口，当有一个接口跨公网调用第三方接口超时时，可能会导致所有接口都不可用，即使大部分接口不依赖于跨公网的第三方调用，也会如此。

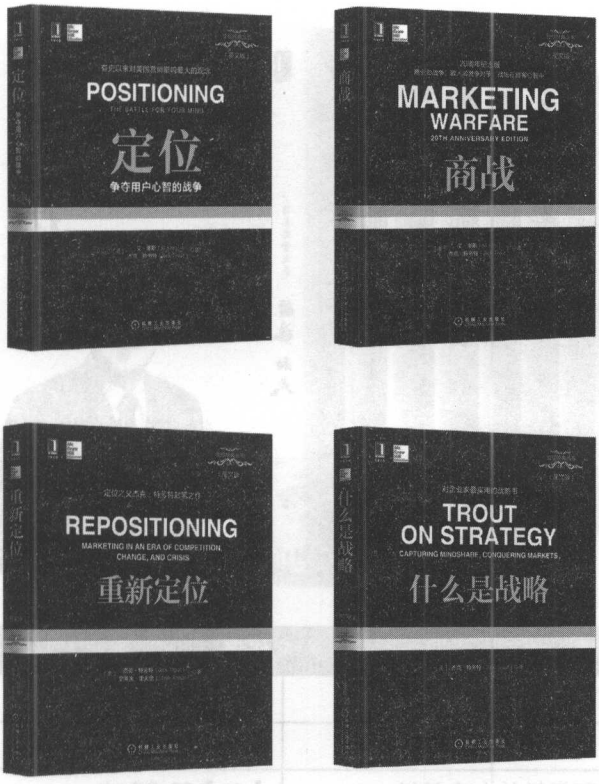
为什么会出现这种情况呢？

内部服务对业务方提供的 N 个接口，会共用服务容器内的工作线程（假设有

100 个工作线程）。假设这 N 个接口的某个接口跨公网依赖于第三方的接口，发生了网络抖动，或者接口超时（不妨设超时时间为 5 秒，潜台词是，这个工作线程会被占用 5 秒钟，然后超时返回业务调用方）；假设这个请求的吞吐量为 20qps，（言下之意，很短的时间内，所有的 100 个工作线程都会被卡在这个第三方超时等待上，而其他 $N-1$ 个原本没有问题的接口，也得不到工作线程处理），则潜在优化方案是什么？

- 增大工作线程数（没有根本解决问题）；
- 降低超时时间（没有根本解决问题）；
- 垂直拆分，N 个接口拆分成若干个服务，使得在出问题时，被牵连的接口尽可能少。（依旧没有根本解决问题，难道一个服务只提供一个接口吗？）

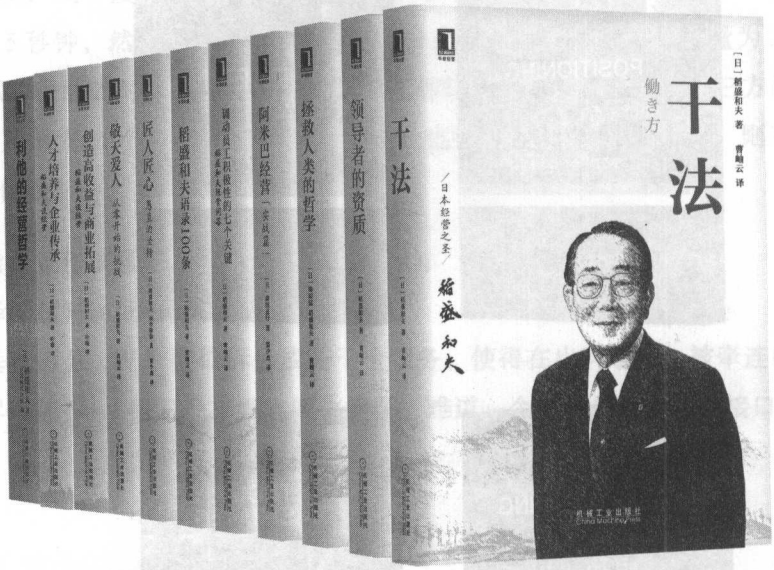
推荐阅读



书名	作者	ISBN	价格
978-7-111-55420-2	定位(英文版)	[美]艾·里斯、杰克·特劳特	89.00
978-7-111-55412-7	商战（英文版）	[美]艾·里斯、杰克·特劳特	89.00
978-7-111-55413-4	重新定位（英文版）	[美]杰克·特劳特、史蒂夫·里夫金	69.00
978-7-111-55208-6	什么是战略（英文版）	[美]杰克·特劳特	69.00
978-7-111-55707-4	简单的力量（英文版）	[美]杰克·特劳特、史蒂夫·里夫金	69.00
978-7-111-55708-1	营销革命（英文版）	[美]艾·里斯、杰克·特劳特	69.00
978-7-111-55882-8	人生定位（英文版）	[美]艾·里斯、杰克·特劳特	69.00

“日本经营之圣”稻盛和夫经营哲学系列

季羡林、张瑞敏、马云、孙正义、俞敏洪、陈春花、杨国安 联袂推荐



ISBN	书名	作者	定价
47025	领导者的资质	【日】稻盛和夫	49.00
49146	稻盛和夫语录100条	【日】稻盛和夫	39.00
48914	调动员工积极性的七个关键	【日】稻盛和夫	45.00
49824	干法	【日】稻盛和夫	39.00
50219	阿米巴经营[实战篇]	【日】森田直行	39.00
51021	拯救人类的哲学	【日】稻盛和夫、梅原猛	39.00
54296	匠人匠心：愚直的坚持	【日】稻盛和夫、山中伸弥	39.00
54638	敬天爱人：从零开始的挑战	【日】稻盛和夫	39.00
57079	赌在技术开发上	【日】稻盛和夫	59.00
57081	企业成长战略	【日】稻盛和夫	49.00
57016	利他的经营哲学	【日】稻盛和夫	49.00
57212	稻盛和夫谈经营：创造高收益与商业拓展	【日】稻盛和夫	45.00
57213	稻盛和夫谈经营：人才培养与企业传承	【日】稻盛和夫	45.00

作者简介

周明耀

海康威视技术高级专家，有超过10年的一线技术团队管理经验。

曾先后就职于瑞士信托银行、美国花旗软件、海康威视研究院。其中，在海康威视研究院主要负责分布式计算领域的系统开发、理论研究，负责的调度系统获得浙江省科技进步奖，独立获得软件著作权，并提交个人发明专利17项。

狂热的技术爱好者，IBM开发者论坛认证专家作者（Java板块中国唯一认证专家），发表科技类文章27篇；InfoQ专栏作家，开设有“技术管理”“冷僻技术”两个专栏。

著有技术畅销书《大话Java性能优化》、《深入理解JVM&G1 GC》。

九三学社社员，九三学社杭州青年工作委员会委员。

维护微信公众号“麦克叔叔每晚10点说”，每晚发布一篇科技类短文。

如何才能带团队，如何才能带出好的团队，是每个程序员都应该思考的问题。带领团队的能力，可能会成为每个程序员晋升的瓶颈。

我们应该都听过某互联网公司爆发的所谓“CTO是否需要写代码”的言语争论事件，我不评论双方对错，不评论这样的指责是否属实，也不评论是否CTO一定需要写代码，我只是觉得，如果是一家成熟的科技公司，它应该从多个维度评审技术团队管理者的工作过程和成绩，而不是采用单一化规则进行评判。

坦白说，技术管理岗位不容易做，既要保证自己的技术说服力，又要经常上一线工作，还要从管理上给予团队支撑，综合这些，对于任何一位程序员来说，都不是那么容易理解和执行的。所以，为了解决技术团队管理者的各种困惑，本书结合作者10余年亲身团队管理经验，分别从技术管理工作、团队创建及人员管理、产品开发过程管理、技术调研/预研、系统架构基础知识五个方面详细展开，还原一个尽责的技术团队管理者需要具备的技术领导力，希望能对你有所帮助。

TECHNICAL LEADERSHIP
FROM PROGRAMMER TO LEADER

上架指导：经管/领导力
ISBN 978-7-111-58914-3
9 787111 589143
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn